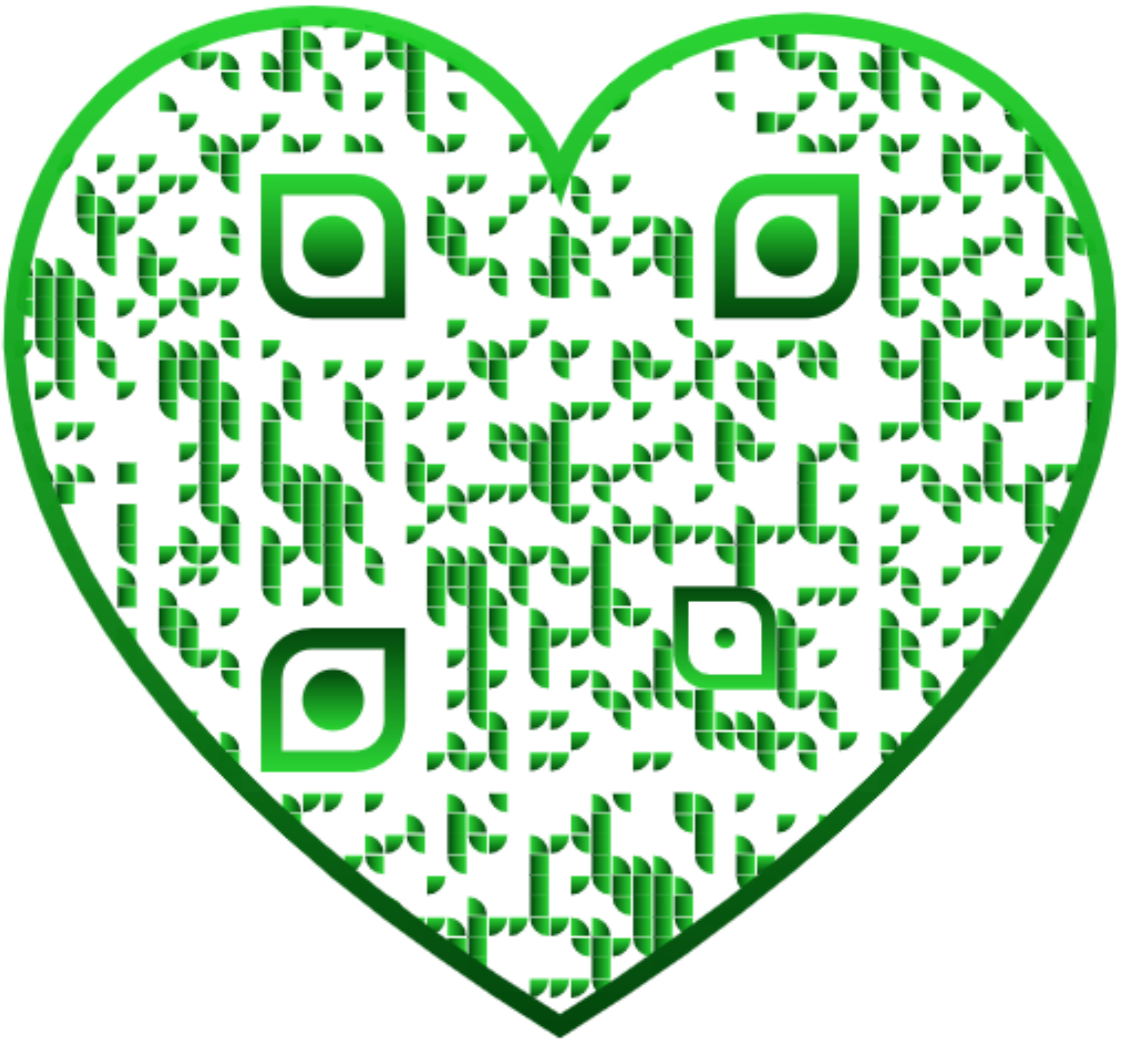
Les bases de données NoSQL et le Big Data

2e édition

Comprendre et mettre en oeuvre

Rudi Bruchez

EYROLLES



Les bases de données NoSQL et le Big Data

2e édition

R. Bruchez

Consultant informatique indépendant, **Rudi Bruchez** est expert en bases de données depuis presque vingt ans (certifications MCDBA et MCITP). Il assure conseil, réalisation, expertise et formation pour la modélisation, l'administration et l'optimisation des serveurs et du code SQL, ainsi que des services autour de SQL Server et des solutions NoSQL. Il est l'auteur ou coauteur de plusieurs ouvrages français et anglais sur SQL Server et SQL, dont *Optimiser SQL Server* (éditions Dunod) et *SQL* (éditions Pearson).

Des bases pour la performance et le Big Data

En quelques années, le volume des données brassées par les entreprises a considérablement augmenté. Émanant de sources diverses (transactions, comportements, réseaux sociaux, géolocalisation...), elles sont souvent structurées autour d'un seul point d'entrée, la clé, et susceptibles de croître très rapidement. Autant de caractéristiques qui les rendent très difficiles à traiter avec des outils classiques de gestion de données. Par ailleurs, l'analyse de grands volumes de données, ce qu'on appelle le Big Data, défie également les moteurs de bases de données traditionnels.

C'est pour répondre à ces différentes problématiques que sont nées les bases de données NoSQL *(Not Only SQL)*, sous l'impulsion de grands acteurs du Web comme Facebook ou Google, qui les avaient développées à l'origine pour leurs besoins propres. Grâce à leur flexibilité et leur souplesse, ces bases non relationnelles permettent en effet de gérer de gros volumes de données hétérogènes sur un ensemble de serveurs de stockage distribués, avec une capacité de montée en charge très élevée. Elles peuvent aussi fournir des accès de paires clé-valeur en mémoire avec une très grande célérité. Réservées jusqu'à peu à une minorité, elles tendent aujourd'hui à se poser en complément du modèle relationnel qui dominait le marché depuis plus de 30 ans.

Du choix de la base NoSQL à sa mise en œuvre

Cet ouvrage d'une grande clarté dresse un panorama complet des bases de données NoSQL, en analysant en toute objectivité leurs avantages et inconvénients. Dans une première partie, il présente les grands principes de ces bases non relationnelles : interface avec le code client, architecture distribuée, paradigme MapReduce, etc. Il détaille ensuite dans une deuxième partie les principales solutions existantes (les solutions de Big Data autour de Hadoop, MongoDB, Cassandra, Couchbase Server...), en précisant spécificités, forces et faiblesses de chacune. Complétée par une étude de cas réel, la dernière partie du livre est consacrée au déploiement concret de ces bases : dans quel cas passer au NoSQL ? quelle base adopter selon ses besoins ? quelles données basculer en NoSQL ? comment mettre en place une telle base ? comment la maintenir et superviser ses performances ?

Au sommaire

Qu'est-ce qu'une base de données NoSQL ? Des SGBD relationnels au NoSQL • Brève histoire des SGBD • Le système dominant : le modèle relationnel d'Edgar Frank Codd • L'émergence du Big Data et des bases NoSQL • **NoSQL versus SQL : quelles différences ?** • Les principes du relationnel en regard du NoSQL • Le transactionnel et la cohérence des données • **Les choix techniques du NoSQL** • L'interface avec le code client • L'architecture distribuée • Le Big Data analytique • **Les schémas de données dans les bases NoSQL** • Schéma implicite • Paires clé-valeur • Bases orientées documents ou colonnes • Documents binaires • Stockage du Big Data analytique • **Panorama des principales bases de données NoSQL. Hadoop et HBase** • Installation et architecture • **Le Big Data analytique** • Sqoop et Hive • Apache Spark • **CouchDB et Couchbase Server** • Mise en œuvre • **MongoDB** • Mise en œuvre et administration • **Riak** • Mise en œuvre et administration • **Redis** • Mise en œuvre • **Cassandra** • Caractéristiques du moteur • Mise en œuvre • **Autres bases** • ElasticSearch • Bases de données orientées graphe • **Mettre en œuvre une base NoSQL. • Quand aller vers le NoSQL et quelle base choisir ? • Mettre en place une solution NoSQL** • Architecture et modélisation • Choisir l'architecture matérielle • Mettre en place la solution et importer les données • Exemples de développements • **Maintenir et superviser une base NoSQL** • Réaliser des tests de charge • Supervision avec les outils Linux, les outils intégrés ou Ganglia • **Étude de cas : le NoSQL chez Skyrock** • Le développement de solutions en interne • Utilisation de Redis • Les applications mobiles : Smax.

À qui s'adresse cet ouvrage ?

– Aux experts en bases de données, architectes logiciels, développeurs...
– Aux chefs de projet qui s'interrogent sur le passage au NoSQL

www.editions-eyrolles.com

EYROLLES

Les bases de données NoSQL et le Big Data

2e édition

Comprendre et mettre en oeuvre

Rudi Bruchez

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

DU MÊME AUTEUR

C. SOUTOU, F. BROUARD, N. SOUQUET et D. BARBARIN. – **SQL Server 2014.**
N°13592, 2015, 890 pages.

C. SOUTOU. – **Programmer avec MySQL (3e édition).**
N°13719, 2013, 520 pages.

C. SOUTOU. – **Modélisation de bases de données (3e édition).**
N°14206, 2015, 352 pages. *À paraître*.

C. SOUTOU. – **SQL pour Oracle (7e édition).**
N°14156, 2015, 666 pages.

R. BIZOÏ – **Oracle 12*c* – Administration.**
N°14056, 2014, 564 pages.

R. BIZOÏ – **Oracle 12*c* – Sauvegarde et restauration.**
N°14057, 2014, 336 pages.

R. BIZOÏ – **SQL pour Oracle 12*c*.**
N°14054, 2014, 416 pages.

R. BIZOÏ – **PL/SQL pour Oracle 12*c*.**
N°14055, 2014, 340 pages.

C. PIERRE DE GEYER et G. PONÇON – **Mémento PHP et SQL (3e édition).**
N°13602, 2014, 14 pages.

R. BIZOÏ – **Oracle 11*g* – Administration.**
N°12899, 2011, 600 pages.

R. BIZOÏ – **Oracle 11*g* – Sauvegarde et restauration.**
N°12899, 2011, 432 pages.

G. BRIARD – **Oracle 10*g* sous Windows.**
N°11707, 2006, 846 pages.

R. BIZOÏ – **SQL pour Oracle 10*g*.**
N°12055, 2006, 650 pages.

G. BRIARD – **Oracle 10*g* sous Windows.**
N°11707, 2006, 846 pages.

G. BRIARD – **Oracle9*i* sous Linux.**
N°11337, 2003, 894 pages.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre Français d'exploitation du droit de copie, 20, rue des Grands Augustins, 75006 Paris.
© Groupe Eyrolles, 2013, 2015, ISBN : 978-2-212-14155-9

Table des matières

AVANT-PROPOS 1
Un ouvrage impartial 1
À propos des exemples de code 3
À propos de la deuxième édition 3

PREMIÈRE PARTIE
Qu'est-ce qu'une base NoSQL ? 5

CHAPITRE 1
Des SGBD relationnels au NoSQL 7
Brève histoire des systèmes de gestion de bases de données 7
Le modèle hiérarchique 8
Codasyl et Cobol 8
Edgard Frank Codd 9
Le système dominant : le modèle relationnel d'Edgar Frank Codd 10
Les règles de Codd 10
De OLTP à OLAP 11
L'émergence du Big Data et des bases NoSQL 13
Hadoop, une implémentation de MapReduce 14
BigTable, encore Google ! 15
L'influence d'Amazon 15
Les évolutions du Big Data 17
Le mouvement NoSQL est-il une avancée ? 18
La nébuleuse NoSQL 19
Tentatives de classement 19
Peut-on trouver des points communs entre les moteurs NoSQL ? 22

CHAPITRE 2
NoSQL versus SQL : quelles différences ? 25
Les principes du relationnel en regard du NoSQL 25

Les structures de données 26
La modélisation 29
Le langage SQL 32
Méfiez-vous des comparaisons 33
Le défaut d'impédance 34
Les NULL 38
Le transactionnel et la cohérence des données 39
Distribution synchrone ou asynchrone 41
Le théorème CAP 43
Journalisation et durabilité de la transaction 47
Big Data et décisionnel 49

CHAPITRE 3
Les choix techniques du NoSQL 55
L'interface avec le code client 55
Les fonctionnalités serveur 57
Les protocoles d'accès aux données 58
L'architecture distribuée 71
La distribution avec maître 72
La distribution sans maître 76
La cohérence finale 87
Le Big Data analytique 97
Le paradigme MapReduce 98
Lisp et les langages fonctionnels 99
Le fonctionnement d'Hadoop MapReduce 100
Hadoop 2 YARN 102
Le Big Data interactif 104

CHAPITRE 4
Les schémas de données dans les bases NoSQL 107
Le schéma implicite 107
Une approche non relationnelle 108
Les paires clé-valeur 109
Les entrepôts clé-valeur 110
Les bases orientées documents 114
JSON 115
Les bases orientées colonnes 117
Les documents binaires 118
Le stockage du Big Data analytique 119
Les contraintes de stockage pour Hadoop 120
Le SequenceFile 121

Le RCFile 122
L'ORC File 123
Parquet 125

DEUXIÈME PARTIE

Panorama des principales bases de données NoSQL . . . 127
Préliminaires 127

CHAPITRE 5

Hadoop et HBase 129
Hadoop 129
Installation 129
Tester l'exécution d'Hadoop 137
HBase 142
Architecture 143
Installation en mode standalone 143
Mise en œuvre 145

CHAPITRE 6

Le Big Data Analytique 149
Présentation 149
Sqoop et Hive 150
Importer les données avec Apache Sqoop 150
Importer les tables dans Hive 154
Gérer les données avec Apache Hue 156
Apache Spark 158
Caractéristiques 159
Architecture 159

CHAPITRE 7

CouchDB et Couchbase Server 161
Présentation de CouchDB 161
Caractéristiques 162
Mise en œuvre de CouchDB 162
Utilisation de Futon 165
Utilisation de l'API REST 165
Utilisation de l'API REST dans le code client 168
Fonctionnalités du serveur 171
Programmation client 176
Présentation de Couchbase Server 178
Caractéristiques 178

Fonctionnalités CouchDB 179
Interface d'administration 180
Accès à Couchbase Server 181
N1QL 182

CHAPITRE 8
MongoDB 185
Présentation 185
Caractéristiques 185
Mise en œuvre 189
L'invite interactive 189
Programmation client 191
Administration 195
Sécurité 196
Montée en charge 196

CHAPITRE 9
Riak 203
Mise en œuvre 203
Utilisation de l'API REST 205
Programmation client 207
Administration 208
Configuration du nœud 208

CHAPITRE 10
Redis 211
Présentation 211
Les types de données 212
Mise en œuvre 213
Installation 213
Configuration 214
Utilisation de redis-cli 214
Exemples d'applications clientes 216
Maintenance 219
Conclusion 224

CHAPITRE 11
Cassandra 225
Caractéristiques du moteur 225
Modèle de données 225
Stockage 226

Mise en œuvre . . . 226
Configuration . . . 227
L'API Thrift . . . 227
Gestion de la cohérence . . . 229
Programmation client . . . 230
Nodetool . . . 232
Conclusion . . . 233

CHAPITRE 12
Les autres bases de données de la mouvance NoSQL . . . 235
ElasticSearch . . . 235
Mise en œuvre . . . 236
Les bases de données orientées graphes . . . 242
Neo4j . . . 242

TROISIÈME PARTIE
Mettre en œuvre une base NoSQL . . . 247

CHAPITRE 13
Quand aller vers le NoSQL et quelle base choisir ? . . . 249
Aller ou non vers le NoSQL . . . 249
Les avantages des moteurs relationnels . . . 251
Que doit-on stocker ? . . . 251
La différence d'approche avec le relationnel . . . 252
Le problème des compétences . . . 252
Quels sont les besoins transactionnels ? . . . 253
Résumé des cas d'utilisation . . . 253
Quelle base choisir ? . . . 256
En résumé . . . 260
Conclusion . . . 261

CHAPITRE 14
Mettre en place une solution NoSQL . . . 263
Architecture et modélisation . . . 263
Stocker du XML . . . 264
Conception pilotée par le domaine . . . 266
La cohérence . . . 267
Design patterns . . . 268
Choisir l'architecture matérielle . . . 271
Évaluer les besoins en mémoire de Redis . . . 272
Évaluer les besoins disque . . . 275

Mettre en place la solution et importer les données 278
Déploiement distribué 278
Outils de gestion de configuration 279
Importer les données 280
Importer du XML 285
Exemples de développements 287

CHAPITRE 15
Maintenir et superviser ses bases NoSQL 289
Réaliser des tests de charge 289
Redis 290
Cassandra 291
Tests génériques 292
Supervision avec les outils Linux 293
La commande iostat 293
La commande pidstat 296
La commande sar 297
Supervision avec les outils intégrés 298
MongoDB 298
Cassandra 299
Redis 300
Supervision avec Ganglia 301
Installer Ganglia 302
Ajouter des modules Python 303

CHAPITRE 16
Étude de cas : le NoSQL chez Skyrock 305
Le développement de solutions en interne 306
Topy 307
Fluxy 307
L'utilisation de Redis 308
Tester la charge 310
Contourner les problèmes 313
Utiliser Redis pour le traitement d'images 314
Les applications mobiles : Smax 314
Exemple de requêtes géographiques dans MongoDB 315

CONCLUSION
Comment se présente le futur ? 317

INDEX 319

Avant-propos

Ce qu'on appelle le « mouvement NoSQL » est encore relativement jeune en France, même s'il existe depuis plusieurs années aux États-Unis. Mais il suscite déjà un vif intérêt et de nombreux débats, plus que ne l'ont fait les précédents écarts par rapport au modèle dominant de la gestion des bases de données, à savoir le modèle relationnel.

L'une des principales raisons de cet intérêt provient de son approche pragmatique. Les moteurs non relationnels regroupés sous la bannière du NoSQL ont été souvent conçus sous l'impulsion d'entreprises qui voulaient répondre à leurs propres besoins. Ils sont également les produits de leur temps, où les machines sont moins chères et la distribution sur plusieurs nœuds est l'une des solutions les plus accessibles de montée en charge.

On pourrait aussi considérer le NoSQL comme le fruit d'une vieille opposition entre le développeur, qui souhaite utiliser ses outils rapidement, de manière presque ludique, et sans aucune contrainte, et l'administrateur, qui protège son système et met en place des mécanismes complexes et contraignants.[1] D'ailleurs, beaucoup de développeurs n'aiment pas les bases de données. Ils sont pourtant obligés de s'en servir, car l'informatique manipule de l'information, laquelle est stockée dans des systèmes de gestion de bases de données (SGBD). Ils vivent comme une intrusion cet élément externe, qui s'intègre mal avec leur code, et qui les oblige à apprendre un langage supplémentaire, faussement facile, le SQL.

Mais le mouvement NoSQL ne peut se résumer à ce rejet du SQL : certains de ses moteurs sont le résultat de recherches et d'expérimentations. D'autres permettent de garantir les meilleures performances possibles en conservant un modèle de données très simple. Sous la bannière NoSQL, ces outils ont ainsi des identités différentes et répondent à des besoins divers, allant du temps réel au *Big Data*.

Un ouvrage impartial

C'est pourquoi écrire un livre sur le NoSQL dans son ensemble pourrait paraître un peu artificiel, puisque en fin de compte, la seule composante qui réunit des outils comme Cassandra, MongoDB ou Redis, c'est l'abandon du modèle relationnel. Mais somme toute, ce n'est pas qu'un petit point commun. Le modèle relationnel est le modèle dominant depuis la fin des années 1980, cela fait

1. Un mouvement, nommé DevOps, essaie d'ailleurs de concilier ces deux points de vue.

plus de trente ans maintenant. S'éloigner de ce modèle comme le font les moteurs NoSQL implique de développer de nouvelles stratégies, de nouveaux algorithmes et de nouvelles pratiques pour la gestion des données. Il y a donc des points communs, des fertilisations croisées que nous allons détailler dans ce livre.

Comme beaucoup de nouvelles technologies qui rencontrent un certain succès, les moteurs NoSQL font l'objet de débats parfois exagérés, où les arguments manquent de précision et d'objectivité. Pour exemple, voici un extrait traduit de l'introduction du livre *Hbase, the Definitive Guide* (Lars George, O'Reilly, 2011), où l'auteur justifie l'utilisation de HBase plutôt qu'un moteur relationnel dans les cas de forte montée en charge :

« La popularité de votre site augmentant, on vous demande d'ajouter de nouvelles fonctionnalités à votre application, ce qui se traduit par plus de requêtes vers votre base de données. Les jointures SQL que vous étiez heureux d'exécuter par le passé ralentissent soudainement et ne se comportent plus de façon satisfaisante pour monter en charge. Vous devrez alors dénormaliser votre schéma. Si les choses continuent à empirer, vous devrez aussi abandonner l'utilisation de procédures stockées, parce qu'elles deviennent trop lentes. En fin de compte, vous réduisez le rôle de votre base de données à un espace de stockage en quelque sorte optimisé pour vos types d'accès.

Votre charge continuant à augmenter et de plus en plus d'utilisateurs s'enregistrant sur votre site, l'étape logique suivante consiste à prématérialiser de temps en temps les requêtes les plus coûteuses, de façon à pouvoir fournir plus rapidement les données à vos clients. Puis vous commencez à supprimer les index secondaires, car leur maintenance devient trop pénible et ralentit la base de données. Vous finissez par exécuter des requêtes qui peuvent utiliser seulement la clé primaire, et rien d'autre. »

Vous avez peut-être maintenant les yeux écarquillés, comme je les ai eus en lisant ce texte. Pour l'auteur, la solution pour optimiser les accès à un moteur relationnel est de dénormaliser, supprimer les jointures, bannir les procédures stockées et enlever les index. Or un moteur relationnel peut justement monter en charge grâce aux optimisations apportées par la normalisation, l'indexation et les procédures stockées qui évitent des allers-retours entre le client et le serveur. Ce genre d'approximations et de contrevérités sur le modèle relationnel est assez commun dans le monde NoSQL. Est-ce que, pour autant, les propos cités ici sont faux ? Pas forcément, cela dépend du contexte et c'est ça qui est important. Dans une utilisation analytique des données, qui force à parcourir une grande partie des tables, les jointures peuvent réellement poser problème, surtout si elles s'exécutent sur un moteur comme MySQL, que beaucoup d'utilisateurs du NoSQL connaissent bien parce qu'ils sont aussi des défenseurs du logiciel libre. MySQL n'avait jusqu'à très récement qu'un seul algorithme de jointure, la boucle imbriquée, qui est inefficace si le nombre de lignes à joindre est important. Quoi qu'il en soit, il est important de préciser de quel contexte on parle pour ne pas induire le lecteur en erreur.

En face, l'incompréhension est également souvent constatée, et le danger qui guette toute discussion sur le sujet est de se transformer en une guerre d'écoles, où les spécialistes et défenseurs du modèle relationnel rejettent avec mépris le mouvement NoSQL, en ignorant les raisons du développement et de l'engouement de ces systèmes qui tiennent en partie aux difficultés et limitations du relationnel.

Tout cela crée des querelles de clocher qui feront des victimes : les utilisateurs et les entreprises qui, eux, recherchent objectivement la meilleure solution de stockage et de gestion de leurs données. Ce livre se propose donc de clarifier la situation, en comparant les deux modèles, en identifiant leurs points forts et leurs faiblesses, mais aussi en essayant de débusquer ce qui se cache derrière les choix de conception des différents moteurs NoSQL. En effet, nous pensons qu'une vision d'ensemble de l'offre de ces nouveaux moteurs vous permettra de vous orienter dans vos choix de développement. Bien entendu, l'approche de cet ouvrage sera également pratique et démonstrative : nous vous expliquerons comment fonctionnent les moteurs NoSQL les plus populaires, pourquoi et comment les choisir, et comment mettre le pied à l'étrier.

À propos des exemples de code

Dans ce livre, nous allons expliquer comment manipuler des bases de données NoSQL, en présentant leurs interfaces, leurs invites de commande, ainsi que des exemples de code client. Pour ces exemples, nous avons choisi d'utiliser Python, principalement parce que c'est un langage très populaire qui compte de nombreux pilotes pour les moteurs NoSQL, et parce que la clarté de sa syntaxe le rend facile à lire et propice à l'apprentissage.

Python, un langage de choix

Peut-être pensez-vous que Python n'est pas un langage d'entreprise au même titre que C++, C# ou Java ? Détrompez-vous. C'est un langage puissant qui est de plus en plus utilisé dans les projets de grande ampleur. Le site codeeval.com, qui permet à ses membres d'évaluer leur compétence en programmation, publie chaque année une estimation de la popularité des langages de programmation, basée sur des tests de codage pour plus de 2 000 entreprises. Le résultat de l'année 2014 (http://blog.codeeval.com/codeevalblog/2014) met ainsi Python au premier rang pour la quatrième année consécutive, avec 30,3% de popularité.

Toutes nos installations et nos exemples seront réalisés sur une machine Linux, dotée de la distribution Ubuntu Server 14.04 LTS. Python y figurant par défaut, il vous restera à installer `pip` *(Python Package Index)*, l'utilitaire de gestion de paquets Python, qui nous servira à récupérer les différents paquets comme les pilotes pour nos moteurs. Voici la procédure d'installation de `pip` sur Ubuntu.

Installez d'abord les paquets suivants à l'aide de `apt-get`, le gestionnaire de paquets de Debian et d'Ubuntu :

```
sudo apt-get install python-pip python-dev build-essential
```

Puis procédez à la mise à jour de `pip` :

```
sudo pip install --upgrade pip
```

À propos de la deuxième édition

Depuis la parution de la première édition de cet ouvrage, le paysage du NoSQL a beaucoup évolué. Cette nouvelle édition a été remaniée en de nombreux points. Nous avons allégé la partie

théorique sur les différences entre SQL et NoSQL, afin de consacrer plus de pages à une considération essentielle : la nature et le mode d'utilisation des bases de données NoSQL. Nous avons également réduit certaines portions de code, notamment celles détaillant l'installation des moteurs NoSQL, pour traiter de sujets plus importants comme les forces et les faiblesses de chacun de ces moteurs, les cas pratiques d'utilisation et leurs principales fonctionnalités. Par ailleurs, nous avons nettement augmenté la place du Big Data car il s'agit du thème dont l'actualité est la plus forte. Les moteurs NoSQL distribués et les méthodes de traitement distribué sont en pleine évolution actuellement et il est capital de comprendre les enjeux de ces développements pour traiter de larges volumes de données.

Partie I

Qu'est-ce qu'une base NoSQL ?

Le terme « NoSQL » a été inventé en 2009 lors d'un événement sur les bases de données distribuées. Le terme est vague, incorrect (certains moteurs NoSQL utilisent des variantes du langage SQL, par exemple Cassandra), mais présente l'avantage d'avoir un effet marketing et polémique certain. Dans cette partie, nous allons aborder les caractéristiques générales des moteurs NoSQL, historiquement, conceptuellement et techniquement, en regard des bases de données relationnelles, mais aussi indépendamment de cette référence.

1

Des SGBD relationnels au NoSQL

Les défenseurs du mouvement NoSQL le présentent comme une évolution bienvenue de l'antique modèle relationnel. Ses détracteurs le considèrent plutôt comme une régression. Le modèle relationnel est apparu dans les années 1970, devenant rapidement le modèle dominant, et jamais détrôné depuis, un peu comme les langages impératifs (comme C++ et Java) dans le domaine de la programmation. Dans ce chapitre, nous allons présenter un bref historique de l'évolution des bases de données informatiques, pour mieux comprendre d'où viennent les modèles en présence, pourquoi ils ont vu le jour et ont ensuite évolué.

Brève histoire des systèmes de gestion de bases de données

Le besoin d'organiser les données, potentiellement de grandes quantités de données, afin d'en optimiser la conservation et la restitution, a toujours été au cœur de l'informatique. La façon dont nous nous représentons l'ordinateur est une métaphore du cerveau humain. Il nous est évident que l'élément central du fonctionnement intellectuel est la mémoire. Sans le stock d'informations que constitue la mémoire humaine, il nous est impossible de produire le moindre raisonnement, car ce dernier manipule des structures, des éléments connus, reconnus et compris, qui proviennent de notre mémoire.

Stocker et retrouver, voilà les défis de la base de données. Ces deux éléments centraux se déclinent bien sûr en différentes sous-fonctions importantes, comme assurer la sécurité ou se protéger des incohérences. Comme nous allons le voir dans ce chapitre, ces sous-fonctions ne font pas

l'unanimité. Depuis toujours, les avis divergent en ce qui concerne les responsabilités assurées par un système de gestion de bases de données.

Ainsi, les premiers défis des SGBD furent simplement techniques et concernaient des fonctions importantes, stocker et retrouver. Un système gérant des données doit être capable de les maintenir sur une mémoire de masse, comme on disait il y a quelques décennies, et doit offrir les fonctions nécessaires pour retrouver ces données, souvent à l'aide d'un langage dédié. Les débuts furent un peu de la même veine que ce que nous voyons aujourd'hui dans le monde des bases NoSQL : une recherche parallèle de différents modèles.

Le modèle hiérarchique

Le modèle hiérarchique est historiquement la première forme de modélisation de données sur un système informatique. Son développement commença dans les années 1950.

On l'appelle « modèle hiérarchique » à cause de la direction des relations qui s'établissent uniquement du parent vers les enfants. Une base de données hiérarchique est composée d'enregistrements *(records)* qui contiennent des champs et qui sont regroupés en types d'enregistrements *(record types)*. Des relations sont créées entre types d'enregistrements parents et types d'enregistrements fils, ce qui forme un arbre hiérarchique. La différence majeure entre ce modèle et le modèle relationnel que nous connaissons maintenant, est la limitation de l'arbre : il ne peut y avoir qu'un seul arbre et les relations ne peuvent se dessiner que du parent vers l'enfant. Les choix de relations sont donc très limités. L'implémentation la plus connue du modèle hiérarchique est le moteur IMS *(Information Management System)* d'IBM, auparavant nommé ICS *(Information Control System)*, qui fut notamment développé dans le but d'assurer la gestion des matériaux de construction du programme spatial de la NASA pour envoyer des hommes sur la Lune dans les années 1960. IMS existe toujours et selon IBM, la meilleure année du moteur en termes de ventes a été 2003.

Codasyl et Cobol

En 1959, le Codasyl (*Conference on Data Systems Languages*, en français Conférence sur les langages de systèmes de traitement de données) conduit à la création d'un consortium dont l'objectif était de développer des standards de gestion de données et ce développement d'un langage d'accès à ces données. Les membres du Codasyl étaient des entreprises, des universitaires et des membres du gouvernement. Ils établirent d'abord un langage d'interrogation de données, le fameux Cobol *(COmmon Business Oriented Language)*. Ce dernier fut développé la même année à l'université de Pennsylvanie. Cobol fait l'objet d'une norme ISO qui, comme la norme SQL, évolua en plusieurs versions. C'est un langage qu'on nomme « navigationnel » : des pointeurs sont posés et maintenus sur une entité, appelée « entité courante », et le pointeur se déplace au gré des besoins vers d'autres entités. Quand on travaille avec plusieurs entités, chaque entité a son pointeur maintenu sur un article. Il s'agit d'une approche purement procédurale à l'accès aux données : parcourir les ensembles d'entités implique d'écrire des boucles (en utilisant des commandes comme `FIND FIRST` et `FIND NEXT`) qui testent des critères et traitent les données ainsi filtrées, par exemple.

Après avoir établi le langage de traitement de données, les membres du Codasyl établirent un standard de structuration des données, qu'on nomme « modèle de données réseau ». Fondamentalement,

il annonce le modèle relationnel. Le modèle réseau permet de décrire des liens entre des articles. Ces liens sont appelés des *sets*, et on peut les considérer, si on veut faire une comparaison avec le modèle relationnel, comme des tables de correspondance entre des articles, propriétaire d'un côté et type membre de l'autre. Les systèmes de gestion de bases de données réseau n'ont pas l'élégance du modèle relationnel, principalement à cause de leur langage de requête, complexe et navigationnel (Cobol).

Edgard Frank Codd

Edgar Frank Codd est un sujet britannique qui étudia les mathématiques et la chimie en Angleterre, et qui se rendit aux États-Unis en 1948 pour travailler comme programmeur pour IBM. Il obtint son doctorat de *Computer Science* à l'université du Michigan. Au milieu des années 1960, il commença à travailler comme chercheur aux laboratoires de recherche d'IBM à San Jose, en Californie. C'est dans ce laboratoire qu'il élabora l'organisation des données selon un modèle basé sur la théorie mathématique des ensembles. Il publia à ce sujet un document interne en 1969, pour exposer sa théorie au sein d'IBM. Malheureusement, les investissements effectués par la société dans le système IMS n'incitèrent pas IBM à s'intéresser à ce modèle différent. Un an plus tard, Codd publia donc un article intitulé *« A Relational Model of Data for Large Shared Data Banks »* dans *Communications of the ACM*, revue de l'association pour la machinerie informatique *(Association for Computing Machinery)*. Cette association à but non lucratif fut fondée en 1947 pour favoriser la recherche et l'innovation. Elle est toujours active aujourd'hui. L'article pose les bases du modèle relationnel en indiquant les bases mathématiques et algébriques de relations. Il est disponible à l'adresse suivante : http://www.acm.org/classics/nov95/toc.html.

Les relations

Le modèle de Codd privilégiait un système de relations basé uniquement sur les valeurs des données, contrairement à d'autres types de relations utilisant des pointeurs sur des entités (modèle réseau), et une manipulation de ces données à l'aide d'un langage de haut niveau implémentant une algèbre relationnelle, sans se préoccuper du stockage physique des données. Cette approche permet d'isoler l'accès aux données de l'implémentation physique, et même de le faire sur plusieurs niveaux à travers le mécanisme des vues, et d'exprimer des requêtes dans un langage beaucoup plus compact que de manière procédurale comme en Cobol. Il s'agit d'un langage déclaratif, algébrique, en quelque sorte fonctionnel, qui ne s'occupe pas des algorithmes informatiques de manipulation des données. A priori, cette pensée est plutôt étrange : comment concevoir qu'un langage de haut niveau, qui ne s'occupe que de décrire les résultats d'opérations sur des données sans jamais spécifier le comment, puisse être efficace. Pour s'assurer des meilleures performances, il faut normalement être au plus proche de la machine. L'histoire du modèle relationnel a montré que cette pensée était erronée dans ce cas : une description correcte de la requête déclarative, alliée à un moteur d'optimisation efficace du côté du serveur permet au contraire d'optimiser automatiquement les performances.

Le développement des moteurs relationnels

En 1974, le laboratoire de San Jose commença à développer un prototype, le System R, pour expérimenter les concepts avancés par Codd. Ils développèrent un langage de manipulation de

données nommé Sequel *(Structured English Query Language)* et un moteur en deux parties, RSS *(Research Storage System)* et RDS *(Relational Data System)*, une séparation utilisée de nos jours dans tous les moteurs relationnels sous forme de moteur de stockage et de moteur relationnel et qui permet effectivement la séparation entre la gestion physique et la gestion logique des données. System R est l'ancêtre de tous les moteurs de bases de données relationnelles contemporains et a introduit un certain nombre de concepts qui constituent encore aujourd'hui la base de ces moteurs, comme l'optimisation de requêtes ou les index en *B-tree* (arbre balancé). Comme Sequel était une marque déposée par une compagnie britannique, le nom du langage fut changé en SQL. En même temps, à l'université de Berkeley, Eugene Wong et Michael Stonebraker s'inspirèrent des travaux de Codd pour bâtir un système nommé Ingres, qui aboutira plus tard à PostgreSQL, Sybase et Informix. De son côté, vers la fin des années 1970, Larry Ellison s'inspira aussi des travaux de Codd et de System R pour développer son moteur de bases de données, Oracle, dont la version 2 fut en 1979 le premier moteur relationnel commercial. IBM transforma System R en un vrai moteur commercial, nommé SQL/DS, qui devint par la suite DB2.

Le système dominant : le modèle relationnel d'Edgar Frank Codd

À partir des années 1980, le modèle relationnel supplanta donc toutes les autres formes de structuration de données. System R avait prouvé qu'un langage déclaratif pouvait s'avérer être un excellent choix pour une interrogation performante des données. Oracle, DB2 et les descendants d'Ingres furent les implémentations qui rendirent les SGBDR populaires auprès des entreprises et des universités.

Le modèle relationnel a développé notre vision de ce qu'est ou doit être une base de données, pour des décennies : séparation logique et physique, langage déclaratif, structuration forte des données, représentation tabulaire, contraintes définies au niveau du moteur, cohérence transactionnelle forte, etc. Ces caractéristiques ont été gravées dans le marbre par Edgar Codd grâce à ce qu'on appelle « les douze règles de Codd ».

> **Les douze règles de Codd**
>
> Ce qu'on appelle les douze règles de Codd sont en fait numérotées de 0 à 12, il y en a donc treize. Une erreur classique de base zéro.

Elles parurent en octobre 1985 dans le magazine *ComputerWorld* dans deux articles maintenant célèbres : *« Is Your DBMS Really Relational? »* et *« Does Your DBMS Run By the Rules? »*.

Les règles de Codd

Vous trouverez ci-après celles qui nous intéressent le plus pour la suite de ce livre.

- Règle 0 – Toutes les fonctionnalités du SGBDR doivent être disponibles à travers le modèle relationnel et le langage d'interrogation.

- Règle 1 – Toutes les données sont représentées par des valeurs présentes dans des colonnes et des lignes de tables.
- Règle 3 – Une cellule peut ne pas contenir de valeur, ou exprimer que la valeur est inconnue, à l'aide du marqueur `NULL`. Il s'agit d'un indicateur spécial, distinct de toute valeur et traité de façon particulière.
- Règle 5 – Le SGBDR doit implémenter un langage relationnel qui supporte des fonctionnalités de manipulation des données et des métadonnées, de définition de contraintes de sécurité et la gestion des transactions.
- Règle 10 – Indépendance d'intégrité : les contraintes d'intégrité doivent être indépendantes des programmes clients et doivent être stockées dans le catalogue du SGBDR. On doit pouvoir modifier ces contraintes sans affecter les programmes clients.
- Règle 11 – Indépendance de distribution : la distribution ou le partitionnement des données ne doivent avoir aucun impact sur les programmes clients.
- Règle 12 – Règle de non-subversion : aucune interface de bas niveau ne doit permettre de contourner les règles édictées. Dans les faits, cette règle implique qu'il n'est possible d'interroger et de manipuler le SGBDR qu'à travers son langage relationnel.

L'ensemble de ces règles indique la voie à suivre pour les systèmes de gestion de bases de données relationnelles. Elles concernent le modèle sous-jacent des bases dites SQL et ne sont jamais totalement implémentées, à cause des difficultés techniques que cela représente.

De OLTP à OLAP

À partir des années 1980, les moteurs relationnels ont donc pris le pas sur les autres systèmes pour les besoins de tous types de données, d'abord pour les systèmes d'entreprises ou d'académies, puis auprès de développeurs indépendants pour des initiatives libres ou personnelles, comme des logiciels *shareware*, des sites web, etc. Même pour des petits besoins, des moteurs embarqués ou locaux comme SQLite (http://www.sqlite.org/) sont largement utilisés.

Assez rapidement, pourtant, un besoin différent s'est manifesté : le modèle relationnel est performant pour une utilisation purement transactionnelle, ce qu'on appelle OLTP *(Online Transactional Processing)*. Une base de données de gestion, par exemple, qu'on utilise dans les PGI (Progiciels de gestion intégrée, ERP en anglais pour *Enterprise Resource Planning*), présente une activité permanente de mises à jour et de lectures de jeux de résultats réduits. On interroge la table des factures filtrées pour un client, ce qui retourne une dizaine de lignes, on requête la table des paiements pour vérifier que ce client est bien solvable, si c'est le cas, on ajoute une facture comportant des lignes de factures, et pour chaque produit ajouté, on décrémente son stock dans la table des produits. Toutes ces opérations ont une envergure limitée dans des tables dont la cardinalité (le nombre de lignes) peut être par ailleurs importante. Mais, par les vertus d'une bonne modélisation des données et grâce à la présence d'index, chacune de ces opérations est optimisée.

Mais qu'en est-il des besoins statistiques ? Comment répondre aux demandes de tableaux de bord, d'analyses historiques voire prédictives ? Dans un PGI, cela peut être une analyse complète des tendances de ventes par catégories de produits, par succursales, par rayons, par mois, par type de clients, sur les cinq dernières années, en calculant les évolutions pour déterminer quelles

catégories de produits évoluent dans quelle région et pour quelle clientèle, etc. Dans ce genre de requête, qu'on appelle OLAP *(Online Analytical Processing)*, qui doit parcourir une grande partie des données pour calculer des agrégats, le modèle relationnel, les optimiseurs de requête des SGBDR et l'indexation ne permettent pas de répondre de manière satisfaisante au besoin.

Le schéma en étoile

Un modèle différent a alors émergé, avec une structuration des données adaptée à de larges parcours de données volumineuses. Ce modèle est orienté autour du schéma en étoile ou en flocon. Faisons la différence entre un modèle normalisé très simple, comme celui reproduit sur la figure 1-1 et un modèle OLAP en étoile, tel que celui représenté sur la figure 1-2.

Figure 1-1
Modèle normalisé OLTP

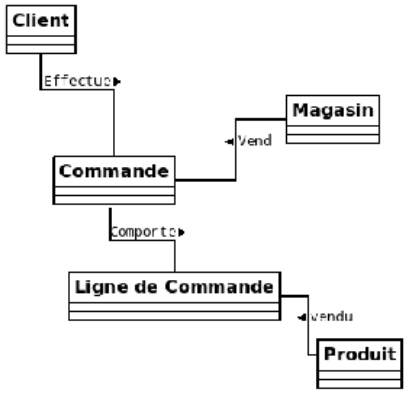
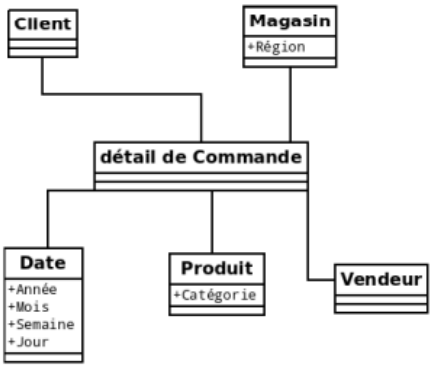


Figure 1-2
Modèle OLAP en étoile



Nous verrons plus loin en quoi consiste la normalisation. Pour l'instant, nous souhaitons simplement montrer comment les structures de données se sont modifiées suite à l'évolution des besoins des entreprises, laquelle dépend aussi de l'augmentation du volume des données manipulées par le système d'information. Le modèle OLAP a vu le jour en raison de l'augmentation du stockage de données agrégées et historiques concernant le fonctionnement et le métier des entreprises, et des besoins de requêtes globales sur ces grands volumes pour des besoins analytiques. C'est ce qu'on appelle le décisionnel ou la *Business Intelligence* : l'utilisation des données pour l'analyse et la connaissance du métier, des caractéristiques commerciales, marketing et comptables de l'entreprise.

Ce modèle, qui a aussi été formalisé par Codd et son équipe, préfigure ce qu'on appelle aujourd'hui le *Big Data*. À la base, l'interrogation du modèle OLAP se fait à travers une vision dite « cube » ou « hypercube », intégrée dans des outils d'analyse, et qui permet de manipuler les données agrégées selon un nombre théoriquement infini de dimensions, qui sont justement les axes d'analyse dont nous parlions. Nous verrons que l'augmentation exponentielle du volume des données aboutit aujourd'hui à chercher de nouvelles façons – ou en tout cas des façons complémentaires – de traiter et d'analyser les données décisionnelles et même celles en temps réel.

L'émergence du Big Data et des bases NoSQL

Les évolutions logicielles suivent assez naturellement les évolutions matérielles. Les premiers SGBD étaient construits autour de mainframes et dépendaient des capacités de stockage de l'époque. Le succès du modèle relationnel est dû non seulement aux qualités du modèle lui-même mais aussi aux optimisations de stockage que permet la réduction de la redondance des données. Avec la généralisation des interconnexions de réseaux, l'augmentation de la bande passante sur Internet et la diminution du coût de machines moyennement puissantes, de nouvelles possibilités ont vu le jour, dans le domaine de l'informatique distribuée et de la virtualisation, par exemple.

Le passage au XXI^e^ siècle a vu les volumes de données manipulées par certaines entreprises ou organismes, notamment ceux en rapport avec Internet, augmenter considérablement. Données scientifiques, réseaux sociaux, opérateurs téléphoniques, bases de données médicales, agences nationales de défense du territoire, indicateurs économiques et sociaux, etc., l'informatisation croissante des traitements en tout genre implique une multiplication exponentielle de ce volume de données qui se compte maintenant en pétaoctets (100 000 téraoctets). C'est ce que les Anglo-Saxons ont appelé le Big Data. La gestion et le traitement de ces volumes de données sont considérés comme un nouveau défi de l'informatique, et les moteurs de bases de données relationnelles traditionnels, hautement transactionnels, semblent totalement dépassés.

La solution Google

Google est probablement la société la plus concernée par la manipulation de grands volumes de données, c'est pourquoi elle a cherché et développé une solution visant à relever ces défis. Google doit non seulement gérer un volume extrêmement important de données afin d'alimenter son moteur de recherche, mais aussi ses différentes offres, comme Google Maps, YouTube, Google Groupes et bien sûr son offre d'outils comme Gmail ou Google Drive. Cela nécessite

non seulement un stockage de données très important, mais aussi des besoins de traitements sur ces volumes.

Afin de permettre le stockage de ces grands volumes de données, Google a développé il y a plus de dix ans un système de fichiers distribué nommé GoogleFS, ou GFS, système propriétaire utilisé seulement chez Google. Fin 2003, au symposium de l'ACM *(Association for Computing Machinery)* sur les principes de systèmes d'exploitation à Lake George, des ingénieurs de Google (Sanjay Ghemawat, Howard Gobioff et Shun-Tak Leung) firent une présentation de Google FS, intitulée *The Google File System* (http://research.google.com/archive/gfs.html). En 2003 déjà, le système de fichiers distribué de Google avait fait ses preuves et était intensivement utilisé en production, ce qui prouvait la viabilité de la solution technique. Le but de GFS était d'offrir un environnement de stockage redondant et résilient fonctionnant sur un *cluster* constitué d'un grand nombre de machines de moyenne puissance, « jetables » (le terme anglo-saxon est *commodity hardware*). Quelques mois plus tard, en 2004, à l'occasion du sixième symposium OSDI *(Operating System Design and Implementation)* à San Francisco, Jeffrey Dean et le même Sanjay Ghemawat de Google présentèrent un autre pilier de leur solution technique. Le titre de la présentation était *MapReduce: Simplified Data Processing on Large Clusters* (http://research.google.com/archive/mapreduce.html). Comme nous le verrons plus en détail, il s'agissait, en plus de stocker les données sur GFS, de pouvoir effectuer des traitements sur ces données de façon également distribuée et de pouvoir en restituer les résultats. Pour ce faire, les ingénieurs de Google se sont inspirés des langages fonctionnels et en ont extrait deux primitives, les fonctions `Map` et `Reduce`. À la base, `Map` permet d'effectuer un traitement sur une liste de valeurs. En le réalisant sur GFS et en regroupant les résultats grâce à la fonction `Reduce`, Google avait réussi à bâtir un environnement de traitement distribué qui lui a permis de résoudre un grand nombre de problèmes.

Hadoop, une implémentation de MapReduce

Doug Cutting, le développeur du moteur de recherche en plein texte Lucene, cherchait un moyen de distribuer le traitement de Lucene pour bâtir le moteur d'indexation web libre Nutch. S'inspirant de la publication sur GFS, il créa avec d'autres contributeurs du projet une implémentation libre en Java nommée d'abord NDFS *(Nutch Distributed File System)*. Afin de permettre un traitement distribué des données accumulées par Nutch, Doug Cutting entama alors une implémentation libre de MapReduce en Java, qu'il appela Hadoop, du nom de l'éléphant doudou de son fils. Nous en reparlerons plus loin ainsi que de sa mise en œuvre. NDFS fut ensuite renommé HDFS *(Hadoop Distributed FileSystem)*. D'abord financé par Yahoo! où Doug Cutting était à l'époque employé, donné à la fondation Apache ensuite, Hadoop est devenu un projet d'une grande popularité, implémenté par de plus en plus de sociétés, jusqu'à Microsoft qui a annoncé début 2012 son soutien au projet, et la disponibilité d'Hadoop sur son service de cloud, Windows Azure, ainsi que sur Windows Server. Au Hadoop Summit 2012, Yahoo! a indiqué que 42 000 serveurs tournaient avec Hadoop, et que leur plus gros cluster Hadoop était composé de 4 000 machines, bientôt 10 000 pour la sortie de Hadoop 2.0. Par ailleurs, en juin 2012, Facebook a annoncé sur son site que leur installation d'HDFS atteignait le volume physique de 100 pétaoctets (http://www.facebook.com/notes/facebook-engineering/under-the-hood-hadoop-distributed-filesystem-reliability-with-namenode-and-avata/10150888759153920).

BigTable, encore Google !

En 2004, Google commença à bâtir un système de gestion de données basé sur GFS : BigTable. En 2006, encore à l'occasion du symposium OSDI, des ingénieurs de Google firent une présentation nommée *BigTable: A Distributed Storage System for Structured Data*. BigTable ressemble à une gigantesque table de hachage distribuée qui incorpore des mécanismes permettant de gérer la cohérence et la distribution des données sur GFS. Une fois de plus, cette présentation de Google inspira la création d'implémentations libres, dont la plus notable est HBase. Cette dernière a été développée à l'origine par la société Powerset, active dans le domaine de la recherche en langage naturel. Leur besoin était de traiter de larges volumes d'informations pour développer leur moteur de recherche. Se basant sur Hadoop et HDFS, ils ajoutèrent la couche correspondant à BigTable. À partir de 2006, deux développeurs de Powerset, Michael Stack et Jim Kellerman, également membres du comité de gestion d'Hadoop, furent dédiés à plein temps au développement de Hbase, projet libre qui commença comme une contribution à Hadoop et devint un projet indépendant de la fondation Apache en janvier 2008. En juillet 2008, Microsoft racheta Powerset pour intégrer leur technologie dans son moteur de recherche Bing. Après deux mois d'inactivité, Microsoft permit aux développeurs de Powerset de continuer leur contribution à plein temps au projet libre.

HBase est devenu depuis un des projets phares du monde NoSQL. Il s'agit d'un système de gestion de données orienté colonnes (voir page 21) destiné à manipuler de très larges volumes de données sur une architecture totalement distribuée. Il est utilisé par des acteurs majeurs du Web, comme Ebay, Yahoo! et Twitter, de même que par des sociétés telles qu'Adobe. Pour consulter une liste non exhaustive des sociétés utilisant Hbase, rendez-vous à l'adresse suivante : http://wiki.apache.org/hadoop/Hbase/PoweredBy.

L'influence d'Amazon

En octobre 2007, Werner Vogels, *Chief Technical Officer* d'Amazon annonce sur son blog, *All Things Distributed* (http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html), qu'Amazon va présenter un papier au 21e symposium sur les principes des systèmes d'exploitation *(Symposium on Operating Systems Principles)* organisé par l'ACM. Cette présentation concerne une technologie qu'Amazon a nommé Dynamo (http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html) et comme celle de Google, sa lecture est très intéressante. Dynamo consiste en un entrepôt de paires clé-valeur destiné à être lui aussi totalement distribué, dans une architecture sans maître. Dynamo utilise un certain nombre de technologies typiques du monde NoSQL que nous détaillerons au chapitre 3 consacré aux choix techniques du NoSQL. Ceci dit, si les moteurs NoSQL implémentent ces technologies, c'est aussi grâce au papier d'Amazon. Cette communication a eu une grande influence dans le monde NoSQL et plusieurs projets se sont basés sur les technologies présentées pour bâtir leur solution libre. Dynamo, comme BigTable de Google, est resté la propriété d'Amazon et le code n'est donc pas disponible.

Un mouvement pragmatique

Voici la traduction d'un passage intéressant de l'article de Werner Vogels :

« Nous avons beaucoup de chance que le papier ait été sélectionné pour publication par le SOSP ; très peu de systèmes réellement utilisés en production ont été présentés durant ces conférences, et de ce point de vue, il s'agit d'une reconnaissance du travail fourni pour créer un système de stockage capable réellement de monter en charge de façon incrémentielle, et dans lequel les propriétés les plus importantes peuvent être configurées de façon appropriée. »

Dans les nouvelles tendances de l'informatique, il y a toujours une part de réponse à des besoins réels de développement, d'invention, de sophistication technique, et une part de mode, d'investissement de grands acteurs du domaine pour développer ou renforcer leur présence, et évidemment pour ne pas se laisser distancer en manquant un rendez-vous avec le futur. Le NoSQL fait partie de ces phénomènes, mais il présente aussi l'avantage d'être un mouvement très pragmatique. Comme nous le verrons, nombre de moteurs NoSQL sont développés par les entreprises elles-mêmes, qui les utilisent en production afin de répondre à leurs propres besoins.

Apache Cassandra

Avinash Lakshman, un des développeurs de Dynamo, fut recruté par Facebook pour créer un entrepôt de données destiné à mettre en place la fonctionnalité de recherche dans les boîtes aux lettres des comptes Facebook, qui permet aux utilisateurs d'effectuer des recherches parmi tous les messages reçus. En collaboration avec un autre ingénieur, Prashant Malik, précédemment Senior Engineer chez Microsoft, il développa un entrepôt orienté colonnes totalement décentralisé, mélangeant donc les technologies présentées par Google et Amazon. Le résultat, Cassandra, fut publié en projet libre sur Google Code en 2008, puis devint un projet Apache en mars 2009. Au fil des versions, Apache Cassandra est devenu un produit très intéressant et représentatif de ce que le mouvement NoSQL peut offrir de meilleur. Sur son site (http://cassandra.apache.org/), on apprend que Cassandra est utilisé par des acteurs comme Twitter, Reddit et Cisco, et que les installations de production Apple représentent 75 000 nœuds et 10 pétaoctets de données, et celles de Netflix, 2 500 nœuds, 420 téraoctets de données et un billion[2] d'opérations par jour.

Ironiquement, lorsque Facebook évalua une nouvelle méthode de stockage pour ses fonctionnalités de messagerie, les tests effectués avec un cluster de machines MySQL, un cluster Cassandra et un cluster HBase les décidèrent à choisir HBase, principalement pour des raisons de facilité de montée en charge et de modèle de cohérence de données (http://www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919). Mais ne vous méprenez pas, ce choix a été fait par Facebook en 2010. Depuis, Cassandra a vécu de très importants changements, aussi bien en termes de performance que d'architecture. Il n'est pas sûr que le même test aujourd'hui produirait les mêmes résultats.

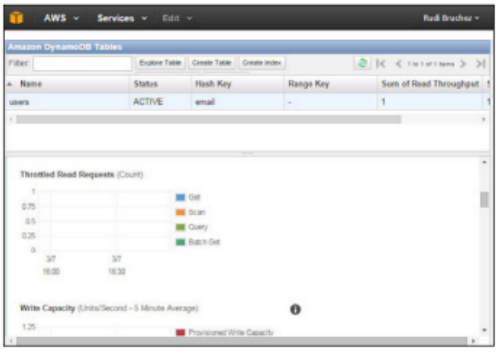
HBase et Cassandra sont des outils à la fois proches et très différents comme nous le verrons plus en détail dans ce livre.

2. Soit un trillion en anglais.

DynamoDB

Amazon Dynamo, nous l'avons vu, a inspiré la création de plusieurs moteurs NoSQL comme Cassandra, Riak ou le projet Voldemort de LinkedIn. Lorsqu'il fut présenté en 2007, Dynamo ne pouvait pas être utilisé directement par d'autres entreprises qu'Amazon. Il était mis en œuvre dans les offres de services web d'Amazon comme S3 *(Simple Storage Service)*, ce qui le rendait indirectement disponible pour les utilisateurs de ces services. Toutefois, en janvier 2012, Amazon a décidé de rendre disponible son moteur à travers une offre cloud nommée DynamoDB (http://aws.amazon.com/fr/dynamodb/), qui n'est pas exactement le même moteur que Dynamo, mais offre une technologie semblable. Un exemple de table géré à travers le service web d'Amazon est représenté sur la figure suivante.

Figure 1-3
Interface d'administration de DynamoDB



Dans un article publié sur son blog en juin 2012 (http://www.allthingsdistributed.com/2012/06/amazon-dynamodb-growth.html), Werner Vogels annonce que de tous les services d'Amazon, DynamoDB est l'offre ayant eu la croissance la plus rapide, et qu'en juin 2012 elle avait déjà dépassé les prévisions d'Amazon pour l'année 2012. DynamoDB permet de gérer des données très rapidement et de monter en charge à travers le cloud d'Amazon sans s'occuper des détails techniques ou matériels. Il met à disposition une API pour accéder aux données à l'aide de langages comme Java, .NET ou PHP. C'est une tendance du NoSQL particulière : l'offre cloud qui représente les avantages de la simplicité et de l'oubli des contraintes matérielles, mais comporte ses désavantages, comme le stockage des données sur des serveurs appartenant à une société tierce, qui plus est américaine, et la dépendance à des services qui ne sont pas exempts de risques de défaillance générale, comme on a pu l'expérimenter quelques fois, même auprès des fournisseurs les plus sérieux. On se souvient notamment qu'en février 2013, Microsoft Azure fut indisponible durant deux jours à cause de l'expiration d'un certificat SSL…

Les évolutions du Big Data

Hadoop a eu un succès fulgurant et il est toujours aujourd'hui la bannière du Big Data. Mais ce n'est qu'une étape, déjà marquée historiquement par la sortie d'une version réarchitecturée

d'Hadoop, appelée Hadoop Yarn ou MapReduce 2 (MRv2). Nous reviendrons sur les détails techniques de ce changement important, l'essentiel étant que la nouvelle architecture devient le socle pour un grand nombre de traitements distribués. Là où la génération précédente d'Hadoop était limitée à un traitement de type Map Reduce en mode batch, Hadoop Yarn permet de diversifier les types de traitement sur de très larges volumes de données, ce qui se révèle notamment très utile de nos jours pour développer des systèmes de traitement en quasi temps réel sur des architectures Big Data. Les outils émergents du Big Data d'aujourd'hui comme Apache Storm ou Apache Spark tournent sur Hadoop Yarn. Nous allons bien sûr parler dans cet ouvrage des importantes évolutions de ces technologies.

Le mouvement NoSQL est-il une avancée ?

On l'a vu dans notre bref historique, l'idée de bases de données non relationnelles n'est pas nouvelle. En fait, le modèle relationnel peut être considéré comme une innovation de rupture qui a supplanté les anciens modèles. Est-ce que pour autant le mouvement NoSQL représente un retour en arrière, comme certains peuvent le dire ? La victoire du relationnel provient de son intelligence et de ses capacités à répondre aux défis du moment, en permettant d'optimiser le stockage et le traitement des données à une époque où les capacités matérielles étaient limitées. Une base de données relationnelle est idéale pour assurer d'excellentes performances et l'intégrité des données sur un serveur dont les ressources sont par définition limitées. Pour réaliser cela, un moteur relationnel offre la capacité de construire un modèle de données, de lui adjoindre des structures physiques comme des index, et d'utiliser un langage d'extraction, le SQL. Cela requiert beaucoup d'efforts et de technicité dans la conception. Les SGBDR sont nés à une époque où les outils devaient être compris et bien utilisés, et où le rythme de développement était plus lent. Cette technicité se perd, les principes et l'implémentation des SGBDR ne s'enseignent pas assez dans les écoles d'informatique, et les développeurs qui s'attaquent à la gestion des données n'ont pas suffisamment conscience des enjeux et des méthodologies spécifiques à ces outils.

De plus, l'avènement de l'informatique distribuée a changé la donne : il ne s'agit plus de faire au mieux par rapport au matériel, mais d'adapter les contraintes logicielles aux possibilités d'extension offertes par la multiplication des machines. Et cela provoque un effet d'entraînement. La distribution permet l'augmentation du volume des données, et l'augmentation du volume des données entraîne un besoin accru de distribution. On sait tous que 90 % du volume de données actuelles a été créé durant les deux dernières années. Une étude d'EMC (http://france.emc.com/index.htm, l'une des grandes entreprises du stockage et du cloud) en 2011 a prévu une multiplication par cinquante du volume de données informatiques entre 2011 et 2020. On n'est évidemment plus du tout dans le domaine d'action des bases de données relationnelles, en tout cas pas des SGBDR traditionnels du marché actuel.

L'une des expressions les plus souvent prononcées par les acteurs du NoSQL est *Web scale*. Un moteur NoSQL est conçu pour répondre à des charges de la taille des besoins du Web. Les besoins ont changé, les types de données manipulées ont aussi évolué dans certains cas, de même que les ressources matérielles.

Les besoins se sont modifiés lorsque les sites web dynamiques ont commencé à servir des millions d'utilisateurs. Par exemple, les moteurs de recherche doivent indexer des centaines de millions de

pages web et offrir des capacités de recherche plein texte sophistiquées et rapides. Ces données sont semi-structurées, et on ne voit pas bien comment on pourrait utiliser un moteur relationnel, qui impose des types de données strictes et atomiques, pour réaliser de telles recherches.

À l'arrivée d'une nouvelle technologie, la tendance humaine est de résister au changement si la technologie actuelle est fermement ancrée, ou au contraire de s'enthousiasmer au-delà du raisonnable si on n'est pas familier avec l'existante, ou si on ne l'aime pas. Le mouvement NoSQL constitue une avancée dès lors que l'on sait en profiter, c'est-à-dire l'utiliser à bon escient, en comprenant sa nature et ses possibilités. C'est justement le but de cet ouvrage. Les capacités de stockage, les besoins de volumétrie, le développement rapide, les approches orientées service, ou encore l'analyse de données volumineuses sont autant de paramètres qui conduisent à l'émergence d'une nouvelle façon, complémentaire, de gérer les données.

La nébuleuse NoSQL

Aujourd'hui, le mot NoSQL est connu de presque tous les informaticiens. Ce mouvement a en effet vécu une nette progression au cours de la dernière décennie. À la parution de la première édition de ce livre, son utilisation était encore réservée à quelques entreprises innovantes. Désormais, le mouvement est bien enclenché, et quelques sociétés de services françaises se spécialisent déjà dans le NoSQL. De nombreuses grandes sociétés, notamment celles présentes sur le Web ou dans le domaine des télécommunications, ou encore certains services publics, s'interrogent et recherchent des techniques leur permettant de monter en charge ou de trouver des solutions à leurs problèmes de traitement de données volumineuses. Nous en verrons un exemple dans le chapitre 16 présentant une étude de cas. Mais, sous la bannière NoSQL, les choix techniques et les acteurs sont divers. Ce n'est pas du tout la même chose que de chercher à traiter en un temps raisonnable des centaines de téraoctets de données, ou de vouloir optimiser un traitement de données en mémoire. Les outils sont donc parfois assez différents. Ils restent néanmoins gouvernés par quelques principes communs, ce qui permet de les regrouper sous le terme de NoSQL.

Tentatives de classement

Avant de voir ce qui regroupe ces outils, essayons d'abord de les distinguer. Quels sont les grands groupes de moteurs NoSQL, et comment les classer ?

Classement par usage

Nous pouvons d'abord tenter de les différencier suivant leur usage, c'est-à-dire selon le type de problématique auquel ils répondent. Nous en avons identifié quatre, que voici.

- **Amélioration des performances** : certains moteurs NoSQL ont pour but d'augmenter au maximum les performances de la manipulation des données, soit pour offrir un espace de cache en mémoire intermédiaire lors du requêtage de SGBDR, soit en tant que SGBD à part entière, qu'il soit distribué ou non. Cet objectif est généralement atteint par trois mécanismes : l'utilisation de la RAM – et nous parlons donc de bases de données en mémoire, la simplification du modèle de données en paires clé-valeur et la distribution du traitement sur les nœuds d'un cluster.

- **Assouplissement de la structure** : pour s'affranchir de la rigidité du modèle relationnel, les moteurs NoSQL simplifient la plupart du temps la structure des données (utilisations de schémas souples comme le JSON, relâchement des contraintes, pas d'intégrité référentielle entre des tables, pas de schéma explicite au niveau du serveur).
- **Structures spécifiques** : certains moteurs NoSQL sont dédiés à des besoins spécifiques, et implémentent donc une structure et des fonctionnalités focalisées sur un cas d'utilisation. Citons par exemple les moteurs orientés graphes, ou les moteurs de recherche plein texte qui offrent également des fonctionnalités proches d'un SGBD, comme Apache Solr ou ElasticSearch.
- **Volumétrie** : l'un des aspects importants des moteurs NoSQL est leur capacité à monter en charge. C'est sans doute même la raison première de la création du mouvement NoSQL. Supporter des volumétries importantes passe par une distribution du stockage et du traitement. C'est une distinction importante que nous approfondirons par la suite. Hadoop est un système de distribution du traitement colocalisé avec un système de distribution de stockage. La distribution du traitement est très importante dans un contexte analytique et dans la plupart des applications Big Data. Le stockage distribué est soit réalisé par des fichiers plats sur un système de fichiers distribués, soit par un moteur de base de données distribué comme Cassandra ou Hbase, conçu pour fonctionner sur un large cluster de machines.

  La distribution peut être centralisée ou décentralisée. Les systèmes centralisés, qui s'inspirent souvent de Google, comportent un maître qui coordonne le stockage et l'aspect transactionnel. Les systèmes décentralisés mettent chaque nœud au même niveau, ce qui implique que chaque nœud comporte un gestionnaire de données et un gestionnaire de cluster, de façon à pouvoir assurer la gestion des données bien sûr, mais aussi pour connaître et maintenir la topologie du cluster, et pouvoir répondre aux requêtes des clients.

La figure suivante représente une carte de ces différents types d'usage.

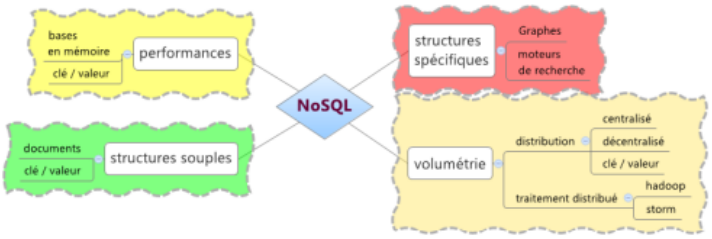


Figure 1-4
Les types d'usage des moteurs NoSQL

Classement par schéma de données

On peut aussi répertorier les moteurs SQL par le schéma ou la structure des données qu'ils manipulent. On distingue dans ce cas cinq modèles.

- **Paires clé-valeur** : les moteurs NoSQL les plus simples manipulent des paires clés valeur, ou des tableaux de hachage, dont l'accès se fait exclusivement par la clé. Il s'agit de moteurs en mémoire comme Redis ou de systèmes distribués inspirés de Dynamo, comme Riak. Ces moteurs offrent des fonctionnalités simplifiées, souvent une moins grande richesse fonctionnelle, en termes de requêtes par exemple, et d'excellentes performances grâce à leur modèle d'accès simplifié. Un système de paires clé-valeur est relativement facile à implémenter : le pattern d'accès par la clé ne nécessite pas de moteur de requête complexe, et ce point d'entrée unique permet de soigner les performances.
- **Les moteurs orientés documents** : le format de sérialisation et d'échange de données le plus populaire est aujourd'hui le JSON *(JavaScript Object Notation)*. Il permet d'exprimer un document structuré qui comporte des types de données simples, mais aussi des listes et des paires clé-valeur, sous une forme hiérarchique. Il est idéal pour représenter des données structurées ou semi-structurées. Il est donc logique de voir apparaître des moteurs dont le format natif de stockage est le JSON. C'est le cas du très populaire MongoDB, mais aussi de CouchDB ou Couchbase Server. La nuance entre moteurs paires clé-valeur et moteurs orientés documents a tendance à s'estomper, parce qu'au fur et à mesure de leur évolution, les premiers intègrent souvent un support du JSON. C'est notamment le cas avec Riak qui depuis sa version 2 supporte l'indexation secondaire, donc sur des documents JSON dans la partie valeur de ses données.
- **Les moteurs orientés colonnes** : inspirés par Google BigTable, plusieurs moteurs NoSQL implémentent une structure proche de la table, dont nous avons déjà parlé et que nous avons définie comme une table de hachage distribuée. Contrairement aux moteurs orientés documents, les données sont ici clairement représentées en lignes et séparées par colonnes. Chaque ligne est identifiée uniquement par une clé, ce qu'on appelle dans le modèle relationnel une clé primaire, et les données de la ligne sont découpées dans des colonnes, ce qui représente un niveau de structuration plus fort que dans les deux modèles précédents. Plus utilisés pour des volumétries importantes, Cassandra ou Hbase appartiennent à cette catégotrie de moteurs.
- **Index inversé** : un index inversé est une correspondance entre un terme, ou du contenu, et sa position dans un ensemble de données, par exemple un document ou une page web. Google utilise un index inversé pour répondre aux recherches sur son moteur. Il existe des moteurs de recherche sur document qu'on apparente au mouvement NoSQL, comme ElasticSearch ou Solr. Ils sont tous deux basés sur Lucene, le moteur d'indexation en index inversé libre. Pourquoi les considérer comme des moteurs de bases de données ? Parce que par-dessus Lucene, ils permettent de manipuler une structure JSON, de la chercher et de la restituer. C'est pratique, parce que cela permet à l'utilisateur de travailler avec une structure de données semblable à un moteur orienté documents, et de profiter d'excellentes capacités de requêtage grâce au moteur de recherche.
- **Structures particulières** : il nous reste juste à regrouper les moteurs NoSQL qui ne rentrent pas dans les catégories précédentes, comme les moteurs orientés graphe, qui utilisent des représentations de données particulières et des méthodes de stockage natives.

Peut-on trouver des points communs entre les moteurs NoSQL ?

Avec l'évolution des moteurs NoSQL, il devient pratiquement impossible de trouver des points communs qui réunissent tous ces moteurs et qui donnent un sens taxinomique au terme NoSQL lui-même. Certaines caractéristiques sont présentes dans presque tous les moteurs mais il demeure toujours des exceptions. Voici quelques-uns de ces points communs.

- **Le schéma implicite**. Alors que dans les moteurs relationnels, le schéma des données doit être prédéfini explicitement au niveau du serveur, dans la quasi-totalité des moteurs NoSQL, le moteur ne contrôle pas le schéma des données et c'est à l'application cliente de structurer correctement ce qu'elle veut manipuler dans la base de données. Nous verrons cela plus en détail dans la suite de l'ouvrage, mais il y a aujourd'hui un contre-exemple de taille : Cassandra qui, à partir de sa version 2 sortie en 2014, impose un schéma fort, prédéfini au niveau du serveur.
- **L'absence de relation**. Que ce soient des tableaux ou des collections, les ensembles de données stockées dans les moteurs NoSQL n'ont pas de relations les unes avec les autres, à l'inverse d'un un moteur relationnel comme son nom l'indique. Vous ne pouvez donc pas créer de référence au niveau du serveur entre un élément d'une collection et plusieurs éléments d'une autre collection. Aucun mécanisme ne le permet. Chaque collection est donc indépendante et c'est un critère important pour assurer une distribution facilitée des données. On peut trouver quelques contre-exemples structurels ou fonctionnels: ainsi, les moteurs orientés graphe implémentent nécessairement des relations entre les nœuds, c'est l'exemple structurel ; et des moteurs de requête pour le Big Data, comme Hive, permettent d'exprimer dans la requête des clauses de jointure, qu'ils vont résoudre derrière le rideau en requêtant les différents ensembles de données, c'est l'exemple fonctionnel.
- **Le langage SQL**. Malheureusement pour le nom NoSQL lui-même, l'absence de langage SQL n'est pas du tout un critère de reconnaissance des moteurs NoSQL. De nombreux moteurs SQL implémentent maintenant un langage déclaratif proche du SQL pour la manipulation des données : CQL en Cassandra, N1QL dans Couchbase Server, par exemple.
- **Le logiciel libre**. Nous l'avons vu, beaucoup de bases NoSQL sont des logiciels libres, et participent au mouvement du libre. Le logiciel libre a maintenant prouvé ses qualités. Une grande partie des serveurs présents sur Internet tournent sur des environnements totalement libres, Linux bien sûr, mais aussi Apache, Tomcat, nginx, MySQL et tant d'autres. Le libre n'est plus synonyme d'amateurisme, et même Microsoft supporte de plus en plus le libre, à travers des développements comme les pilotes PHP et ODBC sur Linux pour SQL Server, le support de Linux sur le cloud Azure, etc. Certaines entreprises comme Facebook, ou pour rester français, Skyrock (voir l'étude de cas chapitre 16), ont la culture du libre et mettent à disposition de la communauté leurs développements internes, sans parler des contributions qu'ils font, financièrement ou en temps de développement aux grands projets libres. Le mouvement NoSQL est donc naturellement un mouvement du logiciel libre. Les éditeurs de SGBD traditionnels, poussés par l'engouement NoSQL, ont des attitudes diverses sur le sujet. Oracle, par exemple, propose un entrepôt de paires clé-valeur distribué nommé simplement Oracle NoSQL Database (http://www.oracle.com/technetwork/products/nosqldb/overview/index.html), avec une version communautaire librement téléchargeable et une version Entreprise qui comportera des améliorations payantes. C'est aussi le choix d'acteurs du monde NoSQL comme Basho, qui développe Riak, ou 10gen avec MongoDB. Des versions Entreprise ou des services additionnels sont payants.

D'autres moteurs sont totalement libres, comme HBase ou Cassandra. Un certain nombre de sociétés se sont créées autour de ces produits pour vendre des services, de l'hébergement ou de la formation, telles que Datastax (http://www.datastax.com/) pour Cassandra ou Cloudera pour la pile Hadoop, HDFS et HBase. Microsoft a annoncé vouloir supporter Hadoop sur Windows et Azure début 2012, notamment en relation avec les fonctionnalités de décisionnel de leur moteur SQL Server. Le service s'appelle maintenant HDInsight (http://www.microsoft.com/sqlserver/en/us/solutions-technologies/business-intelligence/big-data.aspx). On a l'habitude de ces noms très marketing de la part de Microsoft.

Pour toutes ces raisons et pour leur excellente intégration dans les environnements de développement libre, les moteurs NoSQL sont maintenant un choix intéressant de gestion des données pour des projets de développement basés sur des outils libres, dans des langages comme PHP ou Java, et dans tous types d'environnements, également sur les clouds de Microsoft (par exemple, une installation de MongoDB est disponible pour Windows Azure) ou d'Amazon.

2

NoSQL versus SQL : quelles différences ?

S'il y a du NoSQL – qui signifie aujourd'hui, après quelques hésitations, diplomatiquement *Not Only SQL* –, cela veut dire qu'il y a donc aussi du SQL. Quelle est alors la différence entre les deux approches ? Sont-elles si éloignées l'une de l'autre et réellement ennemies ? En quoi le mouvement NoSQL s'écarte-t-il du modèle relationnel et quels sont les choix techniques qui le caractérisent ? Voici les questions auxquelles nous allons tenter de répondre dans ce chapitre.

Les principes du relationnel en regard du NoSQL

Dans le chapitre précédent, nous avons vu d'où venait historiquement le modèle relationnel, et le rôle important qu'a joué Edgar Codd, son créateur, dans l'histoire des bases de données. Voyons maintenant brièvement quelles sont les caractéristiques techniques des SGBDR, ce qui nous permettra de mieux comprendre la différence apportée par le mouvement NoSQL.

Tout d'abord, le modèle relationnel est basé sur le présupposé qu'un modèle mathématique peut servir de base à une représentation des données dans un système informatique. Cela se retrouve dans la conception même des entités que sont les tables des bases de données relationnelles. Elles sont représentées comme des ensembles de données et représentées sous forme de table. Il y a une séparation totale entre l'organisation logique des données et l'implémentation physique du moteur. Les données sont très structurées : on travaille dans le modèle relationnel avec des attributs fortement typés et on modélise la réalité pour la faire entrer dans ce modèle relativement contraignant. Ce système offre l'avantage de pouvoir ensuite appliquer des opérations algébriques

et logiques sur ces données, mais en contrepartie force des données parfois plus complexes à se plier à cette structure prédéfinie.

Cette approche, qui prend sa source dans des modèles théoriques et s'occupe ensuite de l'implémentation physique, est totalement différente du mouvement NoSQL qui se base sur des expérimentations pratiques pour répondre à des besoins concrets. Nous avons vu l'histoire de Dynamo d'Amazon, où le développement a été orienté vers la résolution d'une problématique particulière à l'aide d'un certain nombre de technologies déjà existantes et dont le mariage n'allait pas de soi de prime abord. Beaucoup de moteurs NoSQL utilisent maintenant les technologies de Dynamo parce qu'Amazon a démontré que ces solutions étaient viables pour la simple et bonne raison que cet outil est utilisé avec succès sur le panier d'achat du site Amazon. Il s'agit donc ici plus d'un travail d'ingénieur que d'un travail de chercheur ou de mathématicien.

Les structures de données

Entre le relationnel et le NoSQL, il y a des conceptions fondamentalement différentes de la structure des données. En fait, dans le NoSQL lui-même, il existe différentes façons de considérer la donnée.

Le modèle relationnel se base sur une structuration importante des données. Les métadonnées sont fixées au préalable, les attributs sont fortement typés, et selon les principes édictés par Codd, la normalisation correcte des structures implique qu'il n'y ait aucune redondance des données. Les entités ont entre elles des relations, ce qui permet d'avoir un modèle totalement interdépendant.

Dans le modèle relationnel, la métaphore de la structure, c'est le tableau : une suite de lignes et de colonnes avec sur la première ligne, le nom des colonnes en en-tête. Cela signifie que les métadonnées font partie de l'en-tête et ne sont bien sûr pas répétées ligne par ligne. Cela veut dire également que chaque donnée atomique est stockée dans une cellule, et qu'on peut la retrouver en indiquant le nom de la colonne et l'identifiant de la ligne.

Nous avons vu dans le chapitre précédent les différentes structures de données utilisées dans le monde NoSQL, comme les paires clé-valeur ou les documents JSON.

L'agrégat

Parlons un instant de l'idée d'agrégation. Nous reprenons ici les idées de Pramod J. Sadalage et Martin Fowler dans leur livre *NoSQL Distilled*[3], qui permettent bien de comprendre les différences essentielles de structuration des données entre les bases de données relationnelles et le NoSQL.

Les auteurs soulignent justement que c'est un concept intéressant pour comprendre les structures des bases NoSQL. L'idée provient d'Éric Evans et de son livre *Domain-Driven Design*[4] sur la conception pilotée par le domaine. L'un des concepts de cet ouvrage est l'agrégat, à savoir une collection d'objets liés par une entité racine, nommée la racine de l'agrégat. L'agrégat permet de créer une unité d'information complexe qui est traitée, stockée, échangée de façon atomique. De

3. Pramod J. Sadalage et Martin Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, Addison Wesley, 2012.

4. Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison Wesley, 2003.

l'extérieur, on ne peut faire référence à l'agrégat que par sa racine. En termes de programmation, c'est un concept intéressant parce qu'il permet d'assurer l'unité et l'intégrité d'un bloc d'information en identifiant une racine, on pourrait dire une clé, et en permettant la référence uniquement sur cette clé. C'est une notion qui permet également de modéliser des éléments qu'on trouve dans la réalité. Par exemple, un magazine peut être vu comme une unité, qui contient des pages, des articles, des illustrations, un contenu riche et complexe, mais qui n'existe pas indépendamment du magazine lui-même. Pour y faire référence, vous vous référez au magazine. Pour lire un article, vous devez prendre le magazine et l'ouvrir. La racine représente donc le point d'entrée à ce bloc d'information.

Une base de données orientée documents stocke des données sous forme d'agrégats. Le document JSON contient un bloc d'information, une clé qui permet de l'identifier uniquement dans la collection des documents, et la manipulation du contenu du document se fait à la base en extrayant le document du moteur pour le manipuler dans son intégralité et le stocker à nouveau dans sa totalité.

De ce point de vue, une base de données relationnelle ne structure pas du tout ses données en agrégats. Toute la structure des tables et des relations est à plat, et lorsque nous effectuons une requête SQL, comme celle-ci :

```
SELECT
  c.ContactId, c.Nom, c.Prenom, SUM(f.MontantHT) as TotalFactureHT,
  COUNT(i.InscriptionId) as NbInscriptions
FROM Contact.Contact c
JOIN Inscription.Inscription i ON c.ContactId = i.ContactId
JOIN Inscription.InscriptionFacture inf ON i.InscriptionId = inf.InscriptionId
JOIN Inscription.Facture f ON inf.FactureCd = f.FactureCd
JOIN Stage.Session s ON i.SessionId = s.SessionId
GROUP BY c.ContactId, c.Nom, c.Prenom;
```

nous exprimons dans la requête un agrégat que nous formons à partir des données des tables. Nous prenons la décision de partir d'une table spécifique (dans ce cas, le contact) et nous exprimons par les jointures l'agrégat dont cette table est la racine (ici, on récupère ses inscriptions et le montant de ses factures). On peut donc, à partir d'un modèle relationnel, créer dynamiquement les agrégats souhaités à partir de n'importe quelle racine, de n'importe quel point d'entrée. La requête suivante retourne également le nombre d'inscriptions et le montant des factures, mais cette fois-ci par rapport à une session de formation, donc en partant d'une racine différente.

```
SELECT s.SessionId, s.DateDebut, SUM(f.MontantHT) as TotalFactureHT,
  COUNT(i.InscriptionId) as NbInscriptions
FROM Stage.Session s
JOIN Inscription.Inscription i ON s.SessionId = i.SessionId
JOIN Inscription.InscriptionFacture inf ON i.InscriptionId = inf.InscriptionId
JOIN Inscription.Facture f ON inf.FactureCd = f.FactureCd;
```

L'agrégat est une unité d'information qui va être manipulée par un programme. C'est quelque chose qui sera exprimé par exemple en un objet ou une collection d'objets dans un langage orienté objet.

La centralité de la donnée

Cette souplesse du SGBDR lui permet de servir plusieurs besoins applicatifs. Comme chaque application a besoin d'un agrégat différent, la structure plate d'une base de données relationnelle permet de reconstituer les agrégats demandés par chaque application. Par exemple, une base de données de gestion d'entreprise va permettre de satisfaire les applications des différents services : la logistique, la comptabilité, la relation client, ou la gestion des stocks, qui voudront manipuler les données par des agrégats sur les commandes, les factures, l'identité du client, le produit.

En revanche, si l'agrégat est déjà formé dans la structure de la base de données, comme c'est le cas dans un moteur orienté documents, il sera plus enclin à satisfaire un besoin applicatif, mais pas tous.

Il y a donc dans les bases de données relationnelles une centralité plus forte de la donnée par rapport aux applications, et une seule base de données pourra être le point de convergence de plusieurs processus d'application de l'entreprise. Martin Fowler l'explique de la façon suivante : les bases de données relationnelles sont des outils d'intégration et ils ont été utilisés en développement comme tels. Cette capacité de centralisation a été exploitée par les projets informatiques pour faire converger les diverses applications client vers un seul point.

Est-ce une bonne ou une mauvaise chose ? Un moteur de base de données est-il un outil d'intégration ? En tout cas, on a pris l'habitude de le considérer ainsi, et lorsqu'on entreprend de choisir un moteur de base de données, on pense généralement à un seul moteur pour répondre à tous les besoins.

Si on abandonne l'idée du moteur de base de données unique qui centralise les besoins de toutes les applications, on peut alors choisir différents moteurs selon ses besoins, mais on doit alors considérer la duplication des données, l'échange et la transformation de ces données, bref une couche supplémentaire d'intégration, qu'il faut concevoir, développer, maintenir, avec les contraintes d'architecture, de développement, de performances et bien sûr d'intégrité. On se rapproche de la conception orientée services.

Les structures de données des moteurs NoSQL

La plupart des moteurs NoSQL manipulent des agrégats. Il y a bien sûr des exceptions, et les choses évoluent. Autant vous le dire tout de suite : le monde NoSQL est un terrain en pleine évolution, soyez prêt à tout. Un contre-exemple évident est le groupe des moteurs de bases de données orientés graphe, structurés en nœuds et en relations entre les nœuds. Il s'agit d'outils comme Neo4J. Ils ne sont pas du tout architecturés à partir d'une clé, de la racine d'un agrégat, et ils se rapprochent de ce point de vue des moteurs relationnels. Ils sont également difficiles à distribuer. Cet exemple montre qu'il est difficile de généraliser des caractéristiques dans les moteurs réunis sous la bannière NoSQL.

Nous avons déjà vu deux cas de structure : la structure clé-valeur et la structure orientée documents. Il existe une autre grande catégorie, appelée moteurs orientés colonne. Nous en parlerons plus loin, mais pour schématiser, ces moteurs se basent sur la structure de Google BigTable, et reproduisent une structure en apparence assez proche du modèle relationnel. En tout cas, ils reprennent la métaphore de la ligne et des colonnes. Il y a donc une structure plus formelle, plus contraignante que les autres moteurs. Mais dans les faits, ce que vous stockez dans une colonne

de ces bases de données n'est pas typé, ni limité. Vous avez beaucoup moins de contrainte. Et, concrètement, vous êtes beaucoup plus proche d'un agrégat que d'une table, parce que la clé revêt une plus grande importance. Nous n'en disons pas plus ici, nous entrerons dans les détails au chapitre 4.

Le modèle et la réalité

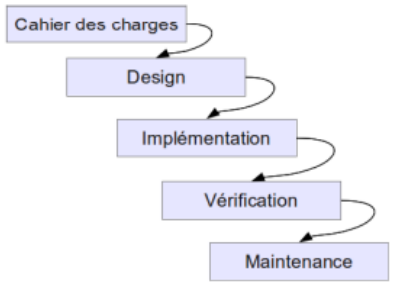
Pour penser et schématiser la réalité, nous avons besoin de modèles : pour penser la matière, nous avons bâti un modèle des atomes qui correspond à de petits systèmes stellaires. Pour penser le découpage le plus élémentaire de la matière, il existe aujourd'hui une théorie des cordes, qui représente ces éléments sous forme de filaments. Cela ne correspond pas à la réalité telle qu'on pourrait l'observer objectivement, c'est simplement un modèle qui va nous permettre, grâce à une simplification et à une adaptation, à quelque chose que nous pouvons saisir cognitivement et représenter. Mais évidemment il ne faut pas confondre réalité et représentation. Et parfois les représentations peuvent changer.

Nous avons déjà signalé qu'un changement de technologie provoque une tension tirant dans deux directions : vers le conservatisme, et vers l'attrait irréfléchi du nouveau. Il y a conservatisme en base de données parce qu'on a pris l'habitude d'un modèle dominant. Le modèle relationnel est juste un modèle, ce n'est pas une vérité objective et naturelle. Il permet beaucoup de choses, mais il n'est pas la vérité révélée et il n'est pas infaillible. Il est facile d'en dénigrer d'autres juste parce qu'on a pris l'habitude d'un modèle. D'autre part, le modèle relationnel possède de nombreuses qualités, et il serait idiot de le remplacer aveuglément par autre chose pour simplement satisfaire un goût de la nouveauté. En fait, le mouvement NoSQL n'est pas une révolution au sens politique du terme : le remplacement d'un ancien régime par un nouveau. On sait en général que ce type de radicalisme s'accompagne d'une forme d'aveuglement. C'est une révolution plutôt copernicienne, dans le sens où à la place d'un modèle hégémonique, qui nous oblige à le choisir en toutes circonstances, on obtient quelque chose de plus riche : le choix.

La modélisation

L'une des caractéristiques les plus importantes du modèle relationnel est justement… les relations, c'est-à-dire le fait de modéliser les données sous forme de tables reliées entre elles par des relations sur des clés. Cela implique une réflexion préliminaire qu'on appelle justement modélisation et dont dépend largement la qualité du système qui sera bâti autour de la base de données. Créer le modèle d'une base de données relationnelle est la première étape du développement, lequel sera d'autant plus facile et efficace que cette étape est réussie. En d'autres termes, pour réussir dans le monde relationnel, il est conseillé de veiller à bien conduire l'étape d'analyse préliminaire. Le modèle relationnel s'intègre bien dans un processus de développement séquentiel tel que le modèle de la chute d'eau *(Waterfall Model)*, illustré sur la figure 2-1.

Figure 2-1
Le modèle Waterfall



Ce modèle bien connu implique que chaque étape du développement logiciel soit maîtrisée et terminée avant de passer à l'étape suivante. C'est pour cette raison que le terme *waterfall* est employé : l'eau coule vers le bas, mais il n'y a jamais de retour aux étapes précédentes. Il est assez clair que ce modèle a montré ses nombreuses limites et qu'il est de plus en plus abandonné de nos jours pour des méthodes dites « agiles », qui considèrent le développement comme un processus itératif, où chaque étape essaie d'être suffisamment courte et souple pour pouvoir corriger rapidement les choix malheureux et être beaucoup plus flexible. L'ironie de ce modèle, utilisé pendant des décennies pour conduire les projets informatiques, est que sa première description formelle peut être trouvée dans un article de Winston Royce intitulé « *Managing the Development of Large Software Systems* » (« Gérer le développement de systèmes logiciels de grande taille »), rédigé pour une conférence ayant eu lieu en 1970 (le papier original est disponible à l'adresse suivante : http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf). Le Waterfall est présenté dans cet article comme un exemple de modèle qui ne fonctionne pas. Quarante ans plus tard, un grand nombre de projets informatiques continuent à être basés sur ce type de procédures.

L'un des reproches faits aux SGBDR par le mouvement NoSQL est donc la rigidité du modèle et les difficultés que pose sa réingénierie. Cela est dû également à la structuration forte des données. Dans un SGBDR, vous devez bâtir autant que possible toute la structure avant de pouvoir travailler avec le contenu. Lorsque le contenu est vivant, il devient ensuite difficile de modifier la structure. Pour cette raison, la plupart des moteurs de bases de données NoSQL appliquent un modèle sans schéma *(schema-less)*, où le schéma des données n'a pas vraiment besoin d'être fixé à l'avance. Nous verrons ce que cela signifie dans le chapitre 3 consacré aux choix techniques des moteurs NoSQL.

La rigidité du modèle relationnel

La modélisation est certes une étape importante. Mais le procès en rigidité qui est fait au modèle relationnel est exagéré. De nos jours, ajouter une colonne dans un moteur relationnel n'est pas très difficile, et l'impact sur l'application cliente est nul si le développeur a bien fait son travail. La rigidité du modèle relationnel est donc relative, et les moteurs NoSQL ne sont pas miraculeusement préservés de ce type de problématique. D'une certaine façon, ce n'est pas la faute du modèle, mais de ses implémentations. La modification d'un modèle relationnel est coûteuse, mais changer des données stockées dans une base NoSQL ne l'est pas forcément moins. Si vous avez stocké des

centaines de milliers de documents JSON qui mentionnent un code de produit, et que la structure de ce code change – on passe par exemple d'un EAN 10 à un EAN 13, comme cela s'est produit sur les ISBN *(International Standard Book Number)* pour le marché du livre –, vous serez bien entendu obligé de modifier vos données comme vous auriez à le faire dans un moteur relationnel. Vous pouvez même avoir plus de travail à faire que dans un moteur relationnel. Pour deux raisons :

- beaucoup de moteurs NoSQL sont plutôt des systèmes de dépôts de données qui n'ont pas de concept de langage de manipulation côté serveur et de traitement efficace des opérations sur des collections de données. Le traitement est souvent unitaire ;
- il n'y a en général pas de mécanisme d'abstraction des structures de données semblables aux vues.

Les choses évoluent et continuent à évoluer. Les moteurs NoSQL sont à la base des outils simplifiés, qui peu à peu s'enrichissent fonctionnellement. Mais pour l'instant le modèle relationnel est plus élégant et plus riche fonctionnellement sur bien des points.

Le partitionnement de données

L'un des problèmes de la normalisation dans un SGBDR concerne la distribution des données et du traitement. Si vous stockez des données qui ont un rapport entre elles, comme des clients, des commandes, des factures, des lignes de facture, etc., dans des tables différentes, vous rencontrerez un problème lorsque vous souhaiterez partitionner ces données, les *sharder*[5]. Vous devez alors vous assurer que les données en rapport les unes avec les autres se trouvent sur le même serveur, en tout cas si vous voulez réaliser des jointures du côté du serveur. Dans le cas contraire, vous devrez effectuer des échanges entre serveurs pour manipuler vos données. De ce point de vue, un stockage orienté documents, qui contient dans un document l'intégralité des données correspondant à la requête à effectuer, quitte à générer une base de données finalement très volumineuse à cause d'un grand nombre de doublons de valeurs, présente un certain nombre d'avantages. Dans un modèle où les données sont découpées, il reste possible de s'assurer que les différentes tables se partitionnent de façon à localiser leurs relations, mais cela implique une gestion manuelle des clés primaires et du *sharding*, ainsi que des techniques spécifiques pour distribuer le plus également possible des données entre nœuds. Ce n'est donc pas une affaire simple et elle est difficile à automatiser.

Toutefois, on ne peut argumenter de façon schématique que le modèle relationnel est non-partitionnable et qu'il faut systématiquement partir sur une solution NoSQL basée sur une agrégation. Les choses évoluent et l'avenir verra sans doute une plus grande tolérance au partitionnement du modèle relationnel. Lorsque nous parlerons du théorème CAP (voir page 43), nous verrons qu'il est impossible d'avoir en même temps une disponibilité satisfaisante, une tolérance au partitionnement et une bonne cohérence des données. La cohérence est une chose, les capacités relationnelles en sont une autre. Il y a aujourd'hui de nombreuses initiatives pour distribuer des moteurs relationnels : plusieurs initiatives tournant autour de MySQL ou de MariaDB comme MySQL Cluster, des bases de données dites NewSQL comme VoltDB ou NuoDB, et le plus

5. Le terme *shard* est souvent utilisé dans les moteurs NoSQL pour exprimer un partitionnement géré automatiquement par le système. Nous en parlerons au chapitre 3.

impressionnant et mystérieux de tous, le système relationnel et transactionnel de Google amené à remplacer son moteur orienté colonnes, Google Spanner.

Google F1

Depuis plusieurs années, Google publie régulièrement des articles sur le développement interne d'un moteur de base de données relationnelle complètement distribuée. Notamment, en 2012, des ingénieurs de l'entreprise ont diffusé un document intitulé « F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business » (http://research.google.com/pubs/pub38125.html) au SIGMOD, le Special Interest Group on Management of Data de l'ACM. Plusieurs systèmes importants de Google étant encore gérés sur MySQL, le géant américain a développé en interne un moteur complètement distribué pour le remplacer. Celui-ci supporte toutes les fonctionnalités importantes d'un SGBDR, avec un moteur de requête parallélisé et la gestion des transactions, tout en s'appuyant sur un stockage complètement distribué. Les mises à jour sont transactionnellement cohérentes, ce qui entraîne une latence plus importante dans les mises à jour, comparé à leur précédent système sur MySQL, mais le document explique quelle a été leur stratégie pour contourner ce problème. Google présente F1 comme le mariage réussi entre les forces du NoSQL (la distribution) et les qualités, notamment transactionnelles, du SGBDR.

En bref, il existe maintenant des implémentations aussi bien que des recherches menées pour permettre le partitionnement d'un modèle basé sur des relations. Aujourd'hui, il demeure beaucoup plus simple de distribuer des données non liées, mais cette situation évoluera peut-être dans un avenir proche.

Le langage SQL

Le langage SQL est un langage d'extraction de données, certes conçu autour du modèle relationnel, mais il n'est pas le modèle relationnel lui-même. SQL est un langage principalement déclaratif, destiné à la manipulation de données et relativement proche du langage naturel. Les bases de données du mouvement NoSQL n'ont pas vocation à s'opposer ou s'éloigner du langage SQL, mais du modèle relationnel. Ce sont les implémentations courantes du modèle relationnel qui comportent les limitations que les bases NoSQL cherchent à dépasser. On devrait donc appeler ce mouvement « NoRelational » plutôt que NoSQL.

Les qualités du langage SQL

Nous ne résisterons pas à citer une conclusion de Philip Greenspun, l'un des pionniers du développement de sites web pilotés par les données, dans un article de blog publié en 2005 et illustrant une requête SQL (http://blogs.law.harvard.edu/philg/2005/03/07/).

« Regardez comme il est amusant de programmer du SQL. Trois lignes de code et vous obtenez une réponse intéressante […]. Comparez cela à du Java et du C, où taper sur votre clavier jusqu'à ce que vos doigts se détachent ne donne en général que peu de résultats. SQL, Lisp et Haskell sont les seuls langages de programmation que j'ai vus où on dépense plus de temps à réfléchir qu'à taper. »

Pourquoi alors vouloir s'éloigner du langage SQL si son expressivité offre un haut niveau d'abstraction et de puissance ? Sans doute pour deux raisons : exprimer une requête correcte en SQL requiert justement plus de réflexion que de programmation. On peut construire une requête SQL

de manière itérative, bloc après bloc, mais ces blocs ne sont pas des opérations, ce sont des éléments de la demande. Bien travailler en SQL demande de ne jamais perdre de vue la question qui est posée, et d'en examiner scrupuleusement chaque mot. C'est réellement une autre façon de penser que celle qui est à l'œuvre dans l'écriture de code impératif. La seconde raison concerne ce qu'on appelle le défaut d'impédance objet-relationnel.

Le requêtage déclaratif en NoSQL

Il y a des évolutions rapides dans le NoSQL, et aussi du point de vue du langage de requête. Depuis la première édition de ce livre, plusieurs changements sont survenus vers l'intégration de langages de requête déclaratifs dans les moteurs NoSQL. Deux exemples : Cassandra, passé en version 2, a remplacé officiellement le langage de l'API Thrift par CQL *(Cassandra Query Language)*, qui reproduit exactement la syntaxe du langage SQL. De son côté, Couchbase Server a introduit dans sa version 3 un langage déclaratif nommé N1QL (prononcer « nickel ») qui utilise une syntaxe `SELECT` très proche du SQL et permet même d'approcher du concept de la jointure avec des instructions comme `NEST` et `UNNEST`.

Autre exemple plus poussé, dans l'analytique interactive avec Hadoop, des outils comme Hive ou Spark proposent une syntaxe SQL qui tente de se rapprocher de la norme SQL. HiveQL inclut ainsi une large part de la syntaxe des jointures définie dans la norme, avec jointures internes, externes, les `CROSS JOIN` et même une syntaxe `SEMI JOIN` pour implémenter dans la clause de jointure la sémantique du mot-clé `EXISTS`. L'une des restrictions de HiveQL est de se limiter à une équijointure : on ne peut écrire des clauses de jointure qui reposent sur autre chose qu'une égalité. Mais c'est de toute façon la grande majorité des cas d'utilisation.

Le terme NoSQL devient donc de moins en moins valide, et dans peu de temps, le seul critère commun subsistant pour essayer de donner une définition générale des moteurs NoSQL, à savoir qu'ils n'utilisaient pas de paradigme relationnel, risque de devenir obsolète.

Méfiez-vous des comparaisons

Une chose assez évidente, mais qui est hélas rarement mise en avant dans les comparaisons faites par les défenseurs du NoSQL avec les moteurs relationnels, c'est que MySQL est souvent utilisé comme point de comparaison. La plupart des entreprises qui ont développé des moteurs NoSQL ont un esprit open source : Google, Facebook, Twitter, LinkedIn utilisent tous intensivement MySQL ou MariaDB. Twitter par exemple a développé en 2011 une couche de sharding sur MySQL nommée Gizzard.

Le problème, c'est que MySQL n'est pas le meilleur moteur relationnel. Il souffre d'un certain nombre de problématiques qui rendent son utilisation intensive et ses performances délicates, et on ne peut pas le prendre comme modèle d'une critique objective de tous les moteurs relationnels. En voici quelques preuves.

Beaucoup de défenseurs du mouvement NoSQL avancent que l'une des forces de ces moteurs est l'absence de jointures, et que ce sont les jointures qui provoquent de forts ralentissements dans les moteurs relationnels. À la base, la jointure n'est pas un problème dans un bon SGBDR. Elle se fait en général entre la clé primaire et une clé étrangère, et si toutes les deux sont indexées, cela équivaut à une recherche de correspondance à travers deux index. Si la cardinalité du résultat

est faible, c'est-à-dire si la correspondance touche un petit nombre de lignes, les algorithmes de jointure des moteurs relationnels offrent d'excellentes performances. En fait, les performances sont meilleures que celles d'un moteur NoSQL, parce que ce qui coûte le plus cher dans un moteur de base de données ce sont les entrées-sorties. À partir du moment où le modèle relationnel élimine la redondance, le volume de données à parcourir pour résoudre la requête est moindre que dans un moteur NoSQL, et donc les performances sont meilleures.

Mais, historiquement, MySQL gère très mal les jointures. Jusqu'à la version 5.6 de MySQL et les versions 5.3 et 5.5 de MariaDB, le seul algorithme disponible est la boucle imbriquée. C'est un algorithme valable sur des petits volumes et qui devient totalement contre-performant sur des volumétries plus importantes. Les moteurs relationnels commerciaux implémentent deux autres algorithmes qui sont la jointure de fusion (merge join ou sort-merge join) et la jointure de hachage (hash join). Notamment, la jointure de hachage est beaucoup plus performante sur des gros volumes.

Donc MySQL n'est vraiment pas un bon point de comparaison pour les jointures volumineuses, car d'autres moteurs relationnels s'en sortent beaucoup mieux. Mais il est clair que lorsqu'on veut effectuer des jointures sur des volumes importants pour des requêtes analytiques où le nombre de correspondances est important, un moteur relationnel va montrer ses limites et le NoSQL arrive à la rescousse.

Autre exemple, on a abordé le sujet de la rigidité du modèle. Si vous utilisez MySQL, vous pouvez facilement avoir l'impression qu'il est très pénible de modifier le schéma de vos tables. Jusqu'à la version 5.6 de MySQL, un certain nombre de modifications de tables étaient effectuées « hors ligne ». Beaucoup de commandes `ALTER TABLE`, au lieu d'effectuer simplement la modification dans la structure de la table, déclenchaient derrière le rideau la création d'une nouvelle table, suivie de la copie de toutes les données de l'ancienne table dans la nouvelle, puis de la suppression de l'ancienne table et enfin du renommage de la nouvelle. Vous imaginez l'effet sur des tables volumineuses dans un environnement fortement multi-utilisateurs. Or, bien entendu, les SGBDR ne fonctionnent pas tous ainsi.

On voit également souvent des données chiffrées de volumétrie. Par exemple, on lit qu'une table comportant un million de lignes deviendrait impossible à gérer dans un moteur relationnel. En réalité, un moteur SQL est capable de gérer correctement des tables qui comportent des centaines de millions de lignes, voire des milliards de lignes, à partir du moment où les recherches qu'on effectue sur ces tables sont sélectives et utilise de bons index. Dans le terme Big Data, Big veut dire vraiment Big.

Le défaut d'impédance

L'une des rugosités du langage SQL est ce que les Anglo-Saxons appellent le défaut d'impédance *(impedance mismatch)* objet-relationnel.

Ce terme est un emprunt à l'électricité. L'impédance est, pour simplifier, une opposition faite par un circuit au passage d'un courant électrique. Optimiser l'impédance consiste à diminuer cette résistance et donc à transférer une énergie de façon optimale. Par défaut d'impédance, les créateurs du terme voulaient signifier que le passage du relationnel à l'objet s'effectue avec une

perte d'énergie et une résistance trop forte. Prenons l'exemple d'un code PHP qui appelle une requête SQL pour faire une recherche dans MySQL :

```
<?php

mysql_connect('localhost','root','********');
mysql_select_db('********');

$nom    = $_REQUEST['nom'];
$prenom = $_REQUEST['prenom'];
$ville  = $_REQUEST['ville'];
$pays   = $_REQUEST['pays'];
$sexe   = $_REQUEST['sexe'];
$age    = $_REQUEST['age'];

$where = "";
$where .= isset($nom)   ? (isset($where)? " OR ": " WHERE " )."( nom    = '$nom') " :
")"
$where .= isset($prenom)? (isset($where)? " OR ": " WHERE " )."( prenom =
'$prenom') ": ")"
$where .= isset($ville) ? (isset($where)? " OR ": " WHERE " )."( ville  = '$ville')
": ")"
$where .= isset($pays)  ? (isset($where)? " OR ": " WHERE " )."( pays   = '$pays')
 ": ")"
$where .= isset($sexe)  ? (isset($where)? " OR ": " WHERE " )."( sexe   = '$sexe')
": ")"
$where .= isset($age)   ? (isset($where)? " OR ": " WHERE " )."( age    = '$age') " :
")"

$sql = "SELECT nom, prenom, ville, pays, sexe, age
FROM Contact
$where
ORDER BY nom, prenom  DESC ";

$req = mysql_query($sql) or die('Erreur SQL!<br />'.$sql.'<br />'.mysql_error());

$data = mysql_fetch_array($req);

mysql_free_result ($req);?>
```

Ici, nous devons construire une chaîne pour bâtir la requête SQL qui sera envoyée telle quelle au serveur. Afin de bâtir la meilleure chaîne possible, nous testons la valeur de chaque paramètre pour éviter d'introduire dans la requête des critères de recherche trop nombreux, ce qui diminuerait les performances de la requête. Le code PHP est reconnu par l'interpréteur, c'est du vrai code. La chaîne SQL, quant à elle, n'est rien d'autre à ce niveau qu'une chaîne. Elle ne sera reconnue que du côté du serveur. Il n'y a pas de réelle interaction entre le code PHP et le code SQL.

Il y a effectivement quelque chose de frustrant que l'on se place d'un côté ou de l'autre de la lorgnette. Si on est un développeur objet, on souffre de devoir sortir du modèle objet pour interroger la base de données. Du point de vue du défenseur du langage SQL, l'élégance de sa syntaxe lui

permet d'exprimer la complexité d'une façon très sobre, et résume des opérations complexes en peu de mots. Les langages objet restent tributaires d'une logique procédurale, où chaque opération doit être exprimée. L'exécution d'un code procédural est aveugle. Pas de raisonnement nécessaire, le code s'exécute comme demandé. Un langage déclaratif comme le SQL est soutenu par un moteur d'optimisation qui décide, en vrai spécialiste, de la meilleure méthodologie pour obtenir les résultats désirés. Il s'agit de deux approches très différentes qui naturellement s'expriment par des langages qui ne fonctionnent pas au même niveau. Cela conduit à rassembler tant bien que mal deux types d'opérations très différentes dans le même code.

Pour traiter ce défaut d'impédance, on peut maquiller l'appel au SQL en l'encapsulant dans des objets qui vont faire office de générateur de code SQL. Les frameworks de mapping objet-relationnel (*Object-Relational Mapping*, ORM) dissimulent ce travail en automatisant la création de classes pour les tables de la base de données, les lignes étant représentées par des instances de ces classes. La théorie des générateurs de code est intéressante, mais en pratique le langage SQL est riche et complexe, et il est impossible d'automatiser élégamment l'écriture d'une requête qui dépasse les besoins élémentaires d'extraction et de modification. Les requêtes générées par des outils comme Hibernate ou Entity Framework sont par la force des choses génériques et grossières, et ne permettent pas de développer toute la puissance du SGBDR. En fait, les requêtes générées sont très souvent bancales et disgracieuses et posent des problèmes de performance. De plus, les ORM déplacent une partie du traitement des données vers le client et se privent ainsi des grandes capacités des moteurs relationnels à manipuler et extraire des ensembles.

Pas de défaut d'impédance en NoSQL ?

Est-ce que les moteurs NoSQL corrigent cette problématique, si tant est qu'elle en soit une ? Oui, d'une certaine façon. Même si certains moteurs NoSQL offrent un langage de requête déclaratif, il est encapsulé dans un pilote adapté à chaque langage client. Voici un exemple de requête vers MongoDB à partir d'un code PHP. Nous essayons de reproduire quelque chose de semblable à notre exemple MySQL.

```
<?
$mongo = new Mongo('mongodb://localhost:27017', array("timeout" =>
2000));
$db = $mongo->selectDB('CRM');

$db->Contact->find(
array('$or' => array(
    array('nom'    => $nom),
    array('prenom' => $prenom),
    array('ville'  => $ville),
    array('pays'   => $pays),
    array('sexe'   => $sexe),
    array('age'    => $age)
)),
array ('nom', 'prenom', 'ville', 'pays', 'sexe', 'age')
);
?>
```

Ici, bien sûr, la requête utilise des objets propres au pilote MongoDB pour PHP développé par MongoDB Inc. Les critères sont envoyés sous forme de double tableau, c'est ensuite à MongoDB de se débrouiller avec ça, par exemple en utilisant un index secondaire s'il est présent. Au passage, la stratégie d'exécution est visible dans MongoDB en utilisant la méthode `explain`, comme ceci :

```
$db->Contact->find(
array('$or' => array(

)),
array ('nom', 'prenom', 'ville', 'pays', 'sexe', 'age')
)->explain();
```

D'autres moteurs, comme CouchDB, offrent une interface REST (*Representational State Transfer*, voir le chapitre 3 consacré aux choix techniques du NoSQL). Envoyer une requête REST et envoyer une requête SQL sont deux actions assez similaires : il faut générer une URI avec des paramètres. La différence est qu'une URI est plus facile à générer qu'une requête SQL, et de nombreuses bibliothèques existent pour tous les langages clients qui permettent d'exprimer les paramètres sous forme de table de hachage ou de tableau, par exemple. Voici un exemple de génération d'une requête REST en Python :

```
import urllib
import urllib2

url = 'http://localhost:5984'
params = urllib.urlencode({
   'nom': nom,
   'prenom': prenom,
   'ville': ville,
   'pays': pays,
   'sexe': sexe,
   'age': age)
})
response = urllib2.urlopen(url, params).read()
```

D'autres moteurs supportent une interface Thrift, qui est une API d'accès que nous détaillerons dans le chapitre 3.

Les ORM pour NoSQL

Les instructions d'extraction des moteurs NoSQL sont simplifiées : on est loin de la complexité du langage SQL. De plus, les performances ne dépendent pas la plupart du temps de la syntaxe des requêtes, mais de la distribution du traitement. Il est donc beaucoup plus facile de concevoir des générateurs de code. Même si ce n'est pas un besoin qui s'est fait sentir avec force, c'est une demande de beaucoup d'équipes de développement habituées à cette manière de travailler, et il existe des bibliothèques pour les moteurs NoSQL les plus importants, par exemple avec MongoDB, mongoose pour Node.JS ou mongoengine pour Python.

Les objets et le relationnel

Une autre réflexion fréquemment rencontrée dans le monde NoSQL comparé au modèle relationnel est le rapport à la « réalité » des données, en tout cas au modèle tel qu'on le bâtit dans l'application cliente, en général sous forme d'objets. L'application cliente organise ses données sous forme d'objets et se trouve obligée de décomposer ces objets pour les stocker dans des tables différentes. Mais les deux modèles sont-ils si différents ? Une bonne organisation des classes permet presque de respecter les trois premières formes normales : chaque objet est une entité distincte dont les propriétés représentent des attributs qui se rapportent strictement à la classe, et les classes entre elles entretiennent des relations avec différentes cardinalités, à travers des collections d'objets. Sérialiser en un document une classe et ses collections d'objets liés dans un document JSON n'est donc pas plus logique que de le faire dans des tables distinctes. Au contraire, l'un des avantages du modèle relationnel est de permettre une grande économie de stockage et une représentation très logique des données, de façon à offrir tous les traitements et toutes les utilisations possibles de ces données. De plus, il n'y a pas de justification logique à conserver les données telles qu'elles sont manipulées par un langage client.

C'est comme si quelqu'un vous disait que les dictionnaires devraient être organisés en phrases complètes, par ordre du sujet, puisque c'est par le sujet que la phrase commence, et ceci parce que tout le monde parle ainsi. Le dictionnaire est organisé par ordre alphabétique de mots, parce que le but d'un dictionnaire est d'y puiser l'un après l'autre les vocables qui composeront les phrases, lesquelles sont de la responsabilité du locuteur, et pas du dictionnaire.

Les NULL

Une autre critique portée aux SGBDR concerne les `NULL`. Du fait de la prédéfinition d'un schéma rigide sous forme de table, ou chaque colonne est prédéfinie, un moteur relationnel doit indiquer d'une façon ou d'une autre l'absence de valeur d'une cellule. Ceci est réalisé grâce au marqueur `NULL`, qui est un animal un peu spécial dans le monde du SGBDR. Au lieu de laisser la cellule vide, on y pose un marqueur, `NULL`, qui signifie valeur inconnue. Le raisonnement est le suivant : laisser une chaîne vide dans la cellule indique une valeur, à savoir « vide ». Le marqueur `NULL` indique clairement l'absence de valeur, ce n'est donc pas une valeur. Ceci dit, selon les implémentations, ce marqueur `NULL` coûte un petit peu en stockage. Une autre problématique du `NULL` est sa gestion compliquée dans le langage SQL. Dans la mesure où il ne s'agit pas d'une valeur, le marqueur `NULL` ne peut pas être comparé, par exemple dans un critère de filtrage (clause `WHERE`) qui est évalué à vrai ou faux. Il impose la gestion d'une logique à trois états : vrai, faux et inconnu. Ainsi, la requête suivante ne retournera pas les lignes dont le titre est `NULL` :

```
SELECT *
FROM Contact
WHERE Titre = 'Mme' OR Titre <> 'Mme';
```

car le `NULL` ne pouvant se comparer, il n'est ni égal à « Mme », ni différent, il est simplement inconnu. Il faut donc chaque fois penser à gérer une logique à trois états de la façon suivante :

```
SELECT *
FROM Contact
WHERE Titre = 'Mme' OR Titre <> 'Mme' OR Titre IS NULL;
```

Les moteurs NoSQL amènent de la souplesse dans cette conception. Dans les moteurs orientés documents ou paires clé-valeur, il n'y a aucune notion de schéma préétabli et de validation d'une structure. Il n'est donc jamais nécessaire d'indiquer un attribut pour dire qu'il est inconnu ! Il suffit de ne pas déclarer l'attribut dans le document. Par exemple, lorsque vous créez un index secondaire sur des éléments du document, cela suppose pour MongoDB que les valeurs des éléments valent `NULL` si elles ne sont pas trouvées dans le document. Il n'y a donc plus besoin de gérer ces valeurs lorsqu'elles n'ont pas de sens dans le document. Cette forme de représentation des données est ainsi beaucoup plus dense que ne l'est celle des SGBDR. Dans les moteurs orientés colonne, les métadonnées des colonnes ne sont pas prédéfinies à la création de la table. En fait, la colonne doit être exprimée à chaque création d'une ligne. Elle peut s'y trouver ou non. On est donc dans le même cas de figure : il n'y a pas réellement de schéma explicite, et les colonnes peuvent être différentes d'une ligne à l'autre. Au lieu de devoir poser un `NULL` en cas d'absence de valeur, il suffit de ne pas exprimer la colonne. Pour cette raison, les SGBD orientés colonnes n'intègrent pas le concept de `NULL`, et on les appelle des entrepôts de données « maigres » (le terme anglais *sparse* est assez difficile à traduire dans ce contexte. On trouve parfois la traduction littérale « éparse », ce qui ne veut pas dire grand-chose).

Problème résolu ?

Donc les moteurs NoSQL résolvent le problème des `NULL` ? Oui, mais il reste deux choses à signaler. D'abord, les `NULL` sont-ils un réel problème ? Ils gênent les requêtes par leur logique à trois états. Dans ce cas, problème résolu. En ce qui concerne l'occupation d'espace des `NULL`, il est minime dans les SGBDR, et on ne peut pas vraiment mettre l'économie d'espace de stockage au crédit des moteurs NoSQL. L'absence de relation entre les tables entraîne en général beaucoup de redondance de données et donc, `NULL` ou pas `NULL`, les moteurs NoSQL occupent beaucoup plus d'espace de stockage qu'une SGBDR au modèle correctement normalisé. Il s'agit donc plus d'un gain en clarté pour la programmation qu'en stockage.

Ironiquement, il nous est arrivé de voir des `NULL` dans les données des moteurs NoSQL, notamment en MongoDB. Non pas que MongoDB le demande, mais simplement parce qu'il s'agissait d'un import des données d'un SGBDR, et que la routine développée pour cet import ne se préoccupait pas de tester si le contenu des colonnes était ou non `NULL` pour générer l'élément dans le document JSON. Le script d'import générait des documents qui comportaient chaque colonne et écrivait le mot « `NULL` » comme valeur de l'élément, le cas échéant.

Le transactionnel et la cohérence des données

Dans le monde informatique, une transaction est une unité d'action qui doit respecter quatre critères, résumés par l'acronyme ACID, et qu'on nomme donc l'acidité de la transaction. Ces critères sont présentés dans le tableau 2-1.

Tableau 2-1. **Critères ACID de la transaction**

Critère	Définition
Atomique	Une transaction représente une unité de travail qui est validée intégralement ou totalement annulée. C'est tout ou rien.
Cohérente	La transaction doit maintenir le système en cohérence par rapport à ses règles fonctionnelles. Durant l'exécution de la transaction, le système peut être temporairement incohérent, mais lorsque la transaction se termine, il doit être cohérent, soit dans un nouvel état si la transaction est validée, soit dans l'état cohérent antérieur si la transaction est annulée.
Isolée	Comme la transaction met temporairement les données qu'elle manipule dans un état incohérent, elle isole ces données des autres transactions de façon à ce qu'elle ne puisse pas lire des données en cours de modification.
Durable	Lorsque la transaction est validée, le nouvel état est durablement inscrit dans le système.

Cohérence peut aussi se dire consistance. La notion de cohérence est importante, et c'est l'un des éléments les plus sensibles entre le monde du relationnel et le monde NoSQL. Les SGBDR imposent une règle stricte : d'un point de vue transactionnel, les lectures de données se feront toujours (dans les niveaux d'isolation par défaut des moteurs) sur des données cohérentes. La base de données est visible en permanence sur un état cohérent. Les modifications en cours sont donc cachées, un peu comme sur la scène d'un théâtre : chaque acte d'une pièce de théâtre se déroule intégralement sous l'œil des spectateurs. Si le décor doit changer, le rideau se baisse, les spectateurs doivent attendre, les changements s'effectuent derrière le rideau et lorsque le rideau se lève, la scène est de nouveau dans un état totalement cohérent. Les spectateurs n'ont jamais eu accès à l'état intermédiaire.

Mais cet état intermédiaire peut durer longtemps, pour deux raisons: plusieurs instructions de modifications de données peuvent être regroupées dans une unité transactionnelle, qui peut donc être complexe. Ensuite, un SGBDR fonctionnant de façon ensembliste, une seule instruction de mise à jour, qui est naturellement et automatiquement transactionnelle, peut très bien déclencher la mise à jour de milliers de lignes de table. Cette modification en masse conservera des verrous d'écriture sur les ressources et écrira dans le journal de transaction ligne par ligne. Tout ceci prend bien évidemment du temps.

Relâchement de la rigueur transactionnelle

Une transaction, c'est donc un mécanisme d'isolation d'une opération simple ou complexe. Elle est importante dans le cadre d'un SGBDR, car du fait du modèle relationnel, une donnée complexe va se décomposer dans plusieurs tables et devra donc faire l'objet de plusieurs opérations d'écriture. Si je considère une donnée comme une commande client, qui comporte les informations du client, des produits commandés et de la facture, il faudra exécuter plusieurs commandes SQL d'insertion, sur les tables client, commande et facture. Le langage SQL ne permet de cibler qu'une table à la fois dans la commande INSERT. L'opération logique de l'insertion d'une donnée complexe devra se traduire par plusieurs opérations physiques d'insertion, et devra être regroupée dans une transaction explicite pour s'assurer qu'il n'est pas possible de se retrouver dans la base de données avec des données partiellement écrites.

On comprend bien qu'un moteur NoSQL, basé sur le principe de l'agrégat, comme nous en avons discuté, ne présente pas ce genre de problème. Dans MongoDB par exemple, on écrit en une seule fois un document JSON qui agrège toutes les informations de la commande. : plus besoin par conséquent d'une transaction explicite. De fait, les critères de la transaction changent de sens : l'écriture est naturellement atomique car le document est écrit en une fois, et elle est naturellement isolée car le document est verrouillé durant l'écriture. Le besoin d'une gestion explicite d'une transaction complexe disparaît, car l'atomicité n'est plus le fait du langage, mais d'une structuration de la donnée décidée par l'architecte ou le développeur.

Il faut donc repenser le concept des critères transactionnels à la lumière d'un système qui gère différemment ses données, et dont les possibilités et les exigences sont autres.

Les systèmes NoSQL agrègent les données, mais quand ils sont distribués, ils les dupliquent. Cela repose le problème transactionnel dans le cadre de la réplication. Si on écrit une donnée plusieurs fois, sur plusieurs nœuds d'un cluster, cette écriture redondante sera-t-elle atomique, cohérente, isolée et durable ? C'est dans ce contexte qu'il faut comprendre les discussions autour de la cohérence *(consistency)* des écritures dans un moteur NoSQL. Nous allons étudier cet aspect plus en détail.

Distribution synchrone ou asynchrone

Nous entrons donc ici brièvement dans des problématiques d'informatique distribuée. La distribution de données peut se concevoir de deux façons : synchrone ou asynchrone. Une distribution synchrone signifie que le processus de réplication des données garantit que les réplicas soient tous alimentés avant de considérer que l'écriture est terminée. Une distribution asynchrone va autoriser la réplication à se faire hors du contrôle du processus de mise à jour. Il n'y aura donc pas de garantie stricte que la réplication s'effectue avec succès.

Le mode de réplication synchrone va permettre de garantir la cohérence des données dans un système distribué, tel que nous venons de le voir. Dans ce cadre, la cohérence est assurée par consensus. Le consensus en informatique, c'est simplement l'accord de plusieurs processus, locaux ou distants, sur une valeur commune. C'est un concept essentiel en calcul et en stockage distribué. Dans l'idéal, tous les processus faisant partie d'un traitement devraient pouvoir participer au consensus, mais il est nécessaire de prendre en compte les risques de défaillance, et donc, un bon protocole permettant d'atteindre un consensus doit être tolérant aux défaillances d'un ou de plusieurs nœuds, pour arriver à former un quorum (une décision à la majorité) plutôt qu'une décision à l'unanimité.

Assurer une réplication à base de consensus en tolérant les pannes n'est possible que sur un réseau synchrone. Cette affirmation a été prouvée dans un article de Fischer, Lynch et Paterson titré « *Impossibility of distributed consensus with one faulty process* »[6]. Cela signifie une chose : si vous voulez assurer une cohérence distribuée, vous devez travailler de façon synchrone. Si vous souhaitez favoriser les performances et travailler de façon asynchrone, vous ne pourrez pas garantir la cohérence de votre réplication. Dans les systèmes asynchrones, on parle donc souvent de cohérence finale *(eventual consistency)* pour exprimer le fait que la cohérence n'est pas garantie

6. Journal of the ACM, vol. 32, n° 2, avril 1985

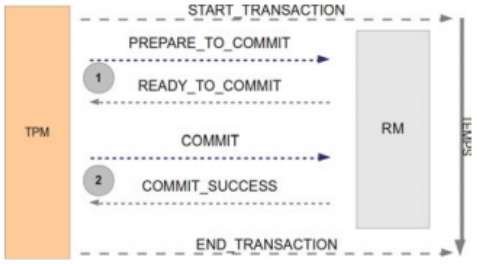
immédiatement mais que des mécanismes vont permettre d'assurer la mise à jour des réplicas de façon asynchrone avec un délai.

La transaction distribuée

Les moteurs relationnels n'ont pas le choix : ils doivent garantir l'acidité de la transaction, donc la cohérence des écritures. Si vous voulez distribuer une base de données relationnelle, vous devez donc être dans un réseau synchrone et utiliser un protocole spécifique qui assure de façon distribuée la cohérence de cette transaction. Il s'agit de ce qu'on appelle une transaction distribuée.

Comme son nom l'indique, une transaction distribuée s'applique sur plusieurs machines. Son implémentation est relativement complexe. Elle s'appuie sur la présence sur chaque machine d'un gestionnaire de ressources (*ressource manager*, RM) qui supporte en général la spécification X/Open XA rédigée par l'*Open Group for Distributed Transaction Processing*. Les machines sont supervisées par un gestionnaire centralisé appelé le *Transaction Processing Monitor* (TPM). Au moment de la validation *(commit)* de la transaction, le TPM organise une validation en deux phases (*two-phase commit*, 2PC), représentée sur la figure suivante.

Figure 2-2
Validation à deux phases dans une transaction distribuée



Lorsque la transaction est prête à être validée, une première phase prépare la validation sur tous les sous-systèmes impliqués, jusqu'à ce que le TPM reçoive un `READY_TO_COMMIT` de la part de tous les RM. Si c'est le cas, la seconde phase de validation réelle peut commencer. Le TPM envoie des ordres de `COMMIT` aux RM et reçoit de leur part des messages de `COMMIT_SUCCESS`. La procédure est donc assez lourde et peut être source de ralentissement dans les systèmes transactionnels qui l'utilisent intensément.

Autre remarque : la validation à deux phases permet de vérifier qu'une transaction peut s'exécuter de façon atomique sur plusieurs serveurs. Il s'agit en fait d'une version spécialisée des protocoles de consensus des systèmes distribués. La première phase de la validation sert à s'assurer que les participants sont prêts à valider, c'est une phase de vote. Si tout le monde est d'accord, la seconde phase réalise la validation. On peut se demander ce qui se passerait si l'un des sous-systèmes venait à crasher après la phase de vote, la seconde phase de validation n'ayant pas encore été réalisée. Dans certains rares cas, cela peut provoquer une erreur de la transaction. Mais dans la plupart des cas, si la machine en erreur redémarre et restaure l'état du moteur transactionnel, ce dernier finira par valider la transaction. Vous voyez peut-être où se situe également le problème : 2PC est un protocole bloquant. La transaction reste ouverte jusqu'à ce que tous les participants aient

envoyé le `COMMIT_SUCCESS`. Si cela dure longtemps, des verrous vont rester posés sur les ressources locales et diminuer la concurrence d'accès.

Paxos

La transaction distribuée est principalement utilisée dans le monde des SGBDR pour la réplication. Elle souffre notamment d'un défaut rédhibitoire qui la rend inutilisable dans le monde NoSQL : l'absence de tolérance aux pannes. Dans la transaction distribuée, si un acteur ne répond pas, la transaction est annulée. Il s'agit en fait d'une forme de consensus à l'unanimité où tous les nœuds doivent être présents. L'objectif d'un moteur NoSQL est d'assurer une montée en charge sur un nombre théoriquement illimité de nœuds, sur du commodity hardware. À tout moment, un nœud peut donc tomber en panne. Il faut par conséquent appliquer un protocole de consensus à la majorité.

Le père des protocoles de consensus s'appelle Paxos. Paxos est un protocole proposé pour la première fois en 1989, notamment par Leslie Lamport, le bien connu créateur de LaTeX. Il existe donc depuis longtemps et a profité d'un certain nombre d'améliorations, notamment en termes de performances, pour répondre au mieux aux besoins dans un environnement distribué moderne. Avec Paxos, vous pouvez maintenir un état partagé et synchrone entre plusieurs nœuds. Pour qu'un tel protocole fonctionne dans le monde réel, il est important qu'il soit tolérant aux pannes. Il est maintenant admis qu'on ne peut bâtir un système informatique distribué sans une forme de résistance aux défaillances matérielles ou de réseaux. Nous verrons dans ce livre que les moteurs NoSQL distribués ont tous des mécanismes de tolérance de panne. C'est un élément nécessaire. Paxos inclut plusieurs mécanismes de tolérance de panne, notamment parce qu'il se base sur une forme de consensus à la majorité et qu'il peut donc continuer son travail même si un ou plusieurs nœuds sont tombés. Si une majorité des nœuds est présente et accepte les modifications, Paxos pourra répliquer les données.

Il n'y a pas d'implémentation officielle de Paxos, il s'agit de la description d'un protocole et les implémentations diverses peuvent varier dans les détails. Une implémentation très utilisée dans les outils qui tournent autour d'Hadoop s'appelle Zookeeper, un gestionnaire de consensus et d'état distribué.

Mais d'un point de vue général, la garantie de la cohérence sur des systèmes distribués est contraignante et entraîne des diminutions de performances en écriture, proportionnellement au nombre de nœuds, alors que l'objectif de la distribution, en tout cas dans le contexte de notre réflexion sur NoSQL, est d'augmenter les performances par une montée en charge horizontale. C'est pour cette raison que l'option de la cohérence finale est choisie dans plusieurs moteurs NoSQL. Le fait que certains moteurs NoSQL ne permettent pas d'assurer la cohérence des données est l'un des sujets majeurs de discorde entre le monde relationnel et le NoSQL. C'est aussi l'un des sujets chauds en raison d'un axiome bien connu dans le monde NoSQL, appelé le théorème CAP.

Le théorème CAP

En juillet 2000, lors du symposium sur les principes de l'informatique distribuée (*Principles of Distributed Computing*, PODC) organisé par l'ACM, le professeur Eric Brewer de l'université de Californie à Berkeley fit une présentation inaugurale intitulée *Towards Robust Distributed*

System. Eric Brewer est non seulement professeur à l'université, mais aussi cofondateur et *Chief Scientist* de la maintenant défunte société Inktomi, que les moins jeunes d'entre nous connaissent comme un des moteurs de recherche web importants de l'ère pré-Google.

Les transparents de cette présentation sont disponibles sur le site d'Eric Brewer : http://www.cs.berkeley.edu/~brewer/. Il s'agit d'une réflexion générale sur la conception de systèmes distribués, ce qui intéresse spécialement dans le monde du NoSQL. Nous pensons qu'il est utile de résumer très brièvement les réflexions d'Eric Brewer, ce qui nous fera nous intéresser au fameux théorème CAP. Classiquement, la focalisation de l'informatique distribuée est ou était faite sur le calcul et non sur les données. C'est pour des besoins de calcul qu'on créait des clusters ou des *grids* de travail, pour atteindre une puissance de calcul multipliée par le nombre de nœuds.

Cluster et grid

On fait en général la différence entre un cluster et un grid de machines de la façon suivante : un cluster est un groupe de machines situé dans le même espace géographique, c'est-à-dire dans le même data center. Un grid est un ensemble de machines distribuées dans plusieurs data centers.

L'exemple tout public de ceci est le projet SETI@home. Ce dernier utilise les machines des utilisateurs volontaires à travers le monde pour analyser les données du projet afin de détecter l'éventuelle présence d'une intelligence extraterrestre (http://setiathome.ssl.berkeley.edu/).

Mais, comme l'affirme Eric Brewer, le calcul distribué est relativement facile. Le paradigme Hadoop le démontre : un algorithme MapReduce, qui distribue le travail et agrège les résultats, n'est pas particulièrement difficile à concevoir et à mettre en œuvre. Ce qui est difficile, c'est la distribution des données. À ce sujet, Brewer énonce le théorème CAP, une abréviation de trois propriétés :

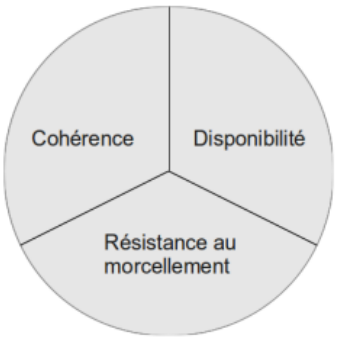
- *consistency* (cohérence) : tous les nœuds sont à jour sur les données au même moment ;
- *availability* (disponibilité) : la perte d'un nœud n'empêche pas le système de fonctionner et de servir l'intégralité des données ;
- *partition tolerance* (résistance au morcellement) : chaque nœud doit pouvoir fonctionner de manière autonome.

Le cas particulier de la résistance au morcellement

L'expression *partition tolerance* n'est pas toujours très claire. On entend par là qu'un système partitionné doit pouvoir survivre aux difficultés du réseau. La définition donnée par Seth Gilbert et Nancy Lynch est la suivante : « Le réseau peut perdre arbitrairement beaucoup de messages envoyés de l'un à l'autre. » Il s'agit ici d'une propriété du réseau. Cela ne signifie donc pas qu'un système soit partitionné, mais cela décrit sa capacité à survivre aux aléas de la perte de partitions ou de communication entre les partitions. Il ne s'agit jamais d'une garantie totale d'accès à partir d'un seul nœud – une situation idéale impossible à atteindre – mais bien d'une capacité à la survie et à la résilience aussi poussée que possible.

Les bases de données partagées ne peuvent satisfaire que deux de ces critères au plus.

Figure 2-3
Le théorème CAP



En 2002, Seth Gilbert et Nancy Lynch du MIT *(Massachusetts Institute of Technology)* ont publié un papier visant à apporter une démonstration de ce principe. Gilbert et Lynch analysent ce trio de contraintes sous l'angle des applications web et concluent qu'il n'est pas possible de réaliser un système qui soit à la fois ACID et distribué.

Si on regarde les bases de données relationnelles traditionnelles selon ce théorème, elles réussissent en matière de cohérence et de support du partitionnement et bien sûr de consistance à travers la garantie ACID des transactions, mais pas en matière de disponibilité.

Cohérence *vs* disponibilité

Si l'objectif d'un système distribué comme une base NoSQL est d'assurer une disponibilité maximale, cela signifie simplement que toute requête envoyée au système NoSQL doit retourner une réponse (le plus rapidement possible, mais cela n'est pas en soi une exigence). Il faut donc pour cela que la perte d'un ou de plusieurs nœuds n'ait pas d'incidence sur les réponses. Dans un système qui privilégie la consistance, on doit s'assurer que les données sont bien lues à partir du ou des nœuds qui contiennent les données les plus fraîches, ou que tous les nœuds sont à jour avant de pouvoir lire la donnée, ce qui pose des problèmes de disponibilité de l'information.

Nous aborderons les solutions techniques apportées par les moteurs NoSQL dans le chapitre suivant. Mais nous pouvons indiquer ici que les solutions des moteurs NoSQL sont assez variées et plus complexes que ce que vous avez peut-être déjà entendu dire. Il ne s'agit pas, pour les moteurs NoSQL asynchrones, d'un abandon pur et simple de la cohérence pour favoriser la disponibilité, les risques sont trop importants. Tout le monde n'est pas prêt à tout miser sur un SGBD distribué qui ne présente aucune garantie de maintien de cohérence de ses données, une sorte de niveau d'isolation chaos.

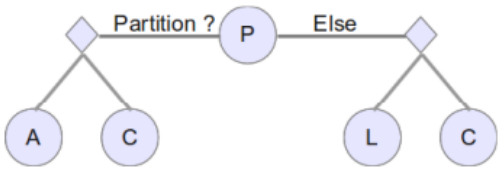
PACELC

Le théorème CAP est encore aujourd'hui sujet de débat. Certains en soulignent les imperfections, d'autres l'affinent en proposant des modèles plus complets, comme le modèle PACELC de Daniel Abadi du département de Computer Science de l'université Yale.

Le modèle PACELC est une sophistication du théorème CAP qui a l'avantage de proposer un cadre d'implémentation plus souple, avec plus de choix. Il nous a semblé intéressant d'en dire un mot pour montrer – nous le verrons en pratique dans la partie II présentant les moteurs NoSQL – que la cohérence n'est pas forcément un critère qu'on respecte scrupuleusement ou qu'on abandonne. Ce n'est pas blanc ou noir. Des moteurs comme Cassandra permettent de décider à chaque instruction du niveau de cohérence et de durabilité. Dans un moteur relationnel, le seul critère sur lequel il est possible d'avoir une action est l'isolation, comme nous l'avons vu. Un moteur comme Cassandra offre plus de souplesse dans la gestion transactionnelle. Nous en reparlerons dans le chapitre 11, dédié à Cassandra.

PACELC est donc un concept de Daniel Abadi. Sa présentation est disponible à l'adresse suivante : http://fr.slideshare.net/abadid/cap-pacelc-and-determinism. Le propos d'Abadi est que le théorème CAP est trop simple, voire simpliste, et qu'il est principalement utilisé comme excuse pour abandonner rapidement la cohérence. Il est facile d'affirmer « Le théorème CAP dit qu'on ne peut pas partitionner, conserver de bonnes performances et maintenir la cohérence des données, donc je laisse tomber la cohérence des données ». Mais, même si on reste dans l'optique du CAP, on peut avoir la disponibilité (A, *Availability*) et la cohérence si l'on n'a pas de partition. Un système peut très bien être partitionné ou non, et donc gérer différemment la cohérence dans ces deux cas. Le modèle PACELC part donc de ce principe et s'énonce comme une forme d'arbre de décisions, représenté sur la figure 2-4.

Figure 2-4
Le modèle PACELC



Le raisonnement est le suivant : si l'on est en présence d'une partition (P), le système va-t-il choisir entre la disponibilité (A) ou la cohérence (C) ? Sinon (E pour *Else*), le système va-t-il choisir entre la latence (L) ou la cohérence (C) ?

Un système comme Dynamo est de type PA/EL : en présence d'une partition, il privilégie la disponibilité au détriment de la cohérence, sinon il choisit la latence. En d'autres termes, il n'a jamais le souci de maintenir la cohérence. De l'autre côté du spectre, un moteur relationnel respectant l'ACIDité de la transaction sera PC/EC : la cohérence sera toujours privilégiée. On se dit donc qu'il manque le meilleur des choix, PA/EC, où la cohérence est privilégiée quand son coût est acceptable. Cette option n'est pas encore réellement mise en place dans les moteurs NoSQL.

Elle l'est dans des moteurs « hybrides » comme GenieDB (http://www.geniedb.com/), un moteur de stockage pour MySQL qui offre une couche de montée en charge horizontale.

Journalisation et durabilité de la transaction

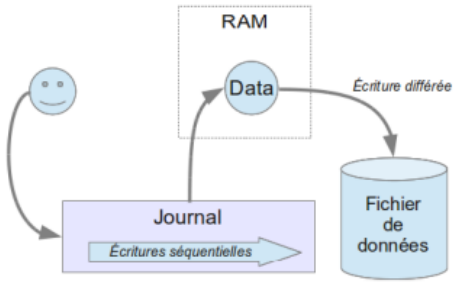
Nous avons vu dans les caractéristiques ACID de la transaction que la durabilité de cette transaction est un élément important. Il est vrai qu'il est plutôt important de ne pas perdre ses données. La durabilité semble a priori quelque chose de naturel : si la modification est écrite quelque part sur le disque, elle est donc durable, en tout cas dans les limites de la solidité du matériel. Pourquoi donc indiquer dans les caractéristiques d'une transaction qu'elle doit être durable ? En fait, dans beaucoup de SGBDR, tout comme dans ce qu'on appelle les bases de données en mémoire dans le monde SQL, les opérations de modification de données s'effectue en mémoire vive, dans un cache de données souvent appelé le *buffer*, et qui évite les écritures directement sur le disque. Les écritures dans des pages de données sont des écritures aléatoires, qui peuvent se produire à n'importe quelle position dans les fichiers de données, et ce type d'écriture est très pénalisant pour les performances.

Amplification d'écritures

Paradoxalement, les effets des écritures aléatoires sont encore plus néfastes sur les disques SSD, à cause d'un phénomène appelé l'amplification des écritures *(write amplification)*. La mémoire flash doit être effacée avant d'être réécrite. Les données sont écrites en unités nommées pages, mais effacées en plus grandes unités, les blocs. Écrire des données signifie donc : lire un bloc, écrire les données encore valides de ce bloc, puis les nouvelles données. Ce n'est pas très efficace. Les disques SSD implémentent souvent un ramasse-miettes pour diminuer ce problème, mais cela implique quand même du déplacement de données. Ensuite, selon les modèles de SSD, une distribution des écritures est réalisée pour éviter que des secteurs soient plus soumis aux écritures que les autres et atteignent prématurément leur limite de modification (une technique appelée *wear leveling*). L'effacement et l'écriture, ainsi que les déplacements éventuels dus aux mécanismes d'écriture de la mémoire flash font que la même écriture peut être physiquement multipliée. Les modèles de SSD affichent un taux d'amplification pour indiquer le nombre de fois où l'écriture est multipliée.

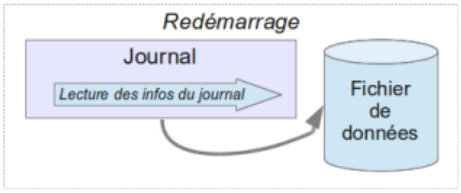
Si, pour éviter les écritures aléatoires, on reste en RAM, donc en mémoire volatile, il est nécessaire de prévoir un mécanisme d'écriture sur le disque pour éviter de perdre ses données en cas d'arrêt de la machine. Le mécanisme choisi pratiquement dans tous les cas est le système du journal. Encore une fois, la traduction française est difficile. Le mécanisme s'appelle *Write-Ahead Logging* (WAL) en anglais, qui pourrait se traduire littéralement par journalisation d'écritures anticipées. Sa représentation schématique est illustrée sur la figure 2-5.

Figure 2-5
Write-Ahead Logging



L'écriture est faite au préalable sur un support de stockage, mais sous forme journalisée, dans un fichier qui écrit séquentiellement les modifications sans se préoccuper de leur allocation finale dans un fichier de données. Il s'agit alors d'une écriture séquentielle, beaucoup moins coûteuse qu'une écriture aléatoire. La donnée est également inscrite dans un cache en mémoire de façon à pouvoir être directement manipulée par le SGBD. Admettons maintenant que la machine tombe. Lorsqu'elle redémarre, le journal est relu et les modifications appliquées dans les données, comme représenté sur la figure 2-6.

Figure 2-6
Phase de redémarrage du WAL



Cette relecture du journal au redémarrage de la base prend du temps, et empêche les données d'être immédiatement accessibles. Afin de limiter cette problématique, une synchronisation du buffer avec les données durables est effectuée à intervalles réguliers, concentrant donc les effets des écritures aléatoires sur une période de temps maîtrisée, souvent configurable.

La technologie du WAL est la plus efficace et a fait ses preuves. Par conséquent, elle est presque universellement utilisée, et les moteurs qui ne l'implémentaient pas et donc n'assuraient pas une réelle durabilité des écritures, l'ont souvent ajoutée au fil des versions. C'est par exemple le cas avec MongoDB dans sa version 1.8 sortie en mars 2011. Depuis la version 2, la journalisation est activée par défaut.

Certains moteurs comme HBase permettent d'écrire directement sur le fichier de données ou de gérer le cache dont nous venons de parler. Cette fonctionnalité s'appelle le *log flushing*, soit le nettoyage du journal en français. Prenons donc HBase comme exemple pour illustrer les différentes options de gestion du journal, qui sont assez communes aux moteurs NoSQL.

L'exemple de HBase

HBase peut écrire ou non dans le WAL. Une propriété booléenne de la classe `org.apache.hadoop.hbase.client.Mutation` dont dérivent les classes `Append` `Delete` et `Put` du client HBase pour Java est nommée `writeToWAL`. Si `writeToWAL` est à `false`, les écritures seront inscrites seulement en mémoire (dans l'équivalent d'un buffer, appelé le *memstore* en HBase). Cette option est à l'évidence une optimisation de performance, mais elle est très risquée. Aucune durabilité n'est assurée et des pertes de données peuvent donc se produire si la machine (le *RegionServer* en HBase) tombe.

En HBase, le comportement par défaut est d'écrire dans le journal, puis d'écrire physiquement la donnée juste après. Le vidage du journal peut aussi être configuré pour être différé. C'est configurable table par table en utilisant les propriétés de celles-ci dans la classe `HTableDescriptor`, qui contient une valeur `DEFERRED_LOG_FLUSH` qui peut être retournée à l'aide de la propriété `isDeferredLogFlush()` et modifiée par la méthode `setDeferredLogFlush(boolean isDeferredLogFlush)`. L'intervalle de vidage peut-être déterminé par l'option suivante du serveur de région : `hbase.regionserver.optionallogflushinterval`. Par défaut, elle est de 1 seconde (1 000 millisecondes).

Big Data et décisionnel

La logique de la croissance de l'informatisation et de la globalisation d'Internet aboutit à une augmentation formidable des volumes de données à stocker et à manipuler. Cette augmentation de volume est de plus imprévisible. Une entreprise fournissant des services ou des biens à des clients régionaux par les canaux de ventes traditionnels que sont par exemple la prospection ou les catalogues papier, peut estimer la croissance de son volume de ventes, et même un succès inespéré restera dans un volume gérable. En revanche, une entreprise fonctionnant sur Internet, qui ne délivre pas de bien manufacturé mais un service qui reste propre à l'environnement du réseau, peut connaître une croissance foudroyante et exponentielle de sa fréquentation. Elle doit être prête à soutenir une très forte augmentation de charge en un rien de temps. Pour cela, une seule solution est envisageable : une montée en charge horizontale, c'est-à-dire que la charge sera distribuée sur des serveurs disposables *(commodity servers)*, ajoutés à une architecture qui permet le plus simplement possible et de façon transparente de multiplier les nœuds.

L'exemple de Zynga

Zynga est une société californienne qui a beaucoup de succès. Elle produit des jeux sur Facebook comme CityVille et FarmVille, qui comptent des dizaines de millions d'utilisateurs. Lorsque Zynga a lancé FarmVille sur Facebook en juin 2009, la société estimait que si elle obtenait 200000 utilisateurs en deux mois, son jeu serait un succès. Mais dès son lancement, le nombre d'utilisateurs augmenta d'un million par semaine, et ce pendant vingt-six semaines d'affilée.

Il est intéressant de noter que cette extraordinaire croissance a été possible même avec un moteur de bases de données relationnelles en stockage disque. L'architecture de Zynga, hébergée en cloud sur Amazon EC2, a pu passer de quelques dizaines de serveurs à plusieurs milliers. Le stockage est assuré dans une base MySQL, mais les données sont principalement cachées en mémoire par memcached, un système de mémoire cache distribué, et c'est là où nous retrouvons du NoSQL.

Dans le monde du Big Data, nous pouvons faire la même distinction que dans le monde des SGBDR concernant l'utilisation des données. Elle a été théorisée en 1993 par Edgar Frank Codd, entre un modèle OLTP et un modèle OLAP.

OLTP et OLAP

Ces modèles représentent des cas d'utilisation des bases de données. OLTP décrit une utilisation vivante et opérationnelle des données. Ces dernières sont en constante évolution, des ajouts, des modifications et des suppressions interviennent à tout moment, et les lectures sont hautement sélectives et retournent des jeux de résultats limités. Les bases de données constituent le socle des opérations quotidiennes des entreprises. Si on transpose le besoin dans le monde NoSQL, on peut dire que les traitements OLTP nécessitent une latence plus faible, donc de meilleurs temps de réponse, avec potentiellement un volume de données plus réduit. OLAP, ou traitement analytique en ligne, décrit une utilisation analytique, décisionnelle des données. Il s'agit de volumes importants de données historiques qui représentent toutes les données de l'entreprise, et qui sont requêtées afin d'obtenir des informations agrégées et statistiques de l'activité de l'entreprise (décisionnel, ou *Business Intelligence*), ou pour extraire des informations nouvelles de ces données existantes à l'aide d'algorithmes de traitement des données *(Data Mining)*. Dans le monde NoSQL, cette utilisation correspond à des besoins de manipulation de grands volumes de données, sans avoir nécessairement besoin de pouvoir les travailler en temps réel. En d'autres termes, une certaine latence est acceptable.

Ces deux modes d'utilisation des données ont des exigences et des défis différents. Un système de gestion de bases de données relationnelles est bien adapté à une utilisation OLTP. Lorsqu'une base relationnelle est bien modélisée, elle permet une grande efficacité des mises à jour grâce à l'absence de duplication des données, et une grande efficacité dans les lectures grâce à la diminution de la surface des données. Jusqu'à un volume même respectable, un bon SGBDR offre d'excellentes performances, si l'indexation est bien faite.

Les exigences de l'OLAP

En revanche, le modèle relationnel n'est pas adapté à une utilisation OLAP des données car dans ce cas de figure, toutes les opérations sont massives: écritures en masse par un outil d'ETL *(Extract, Transfer, Load)*, les lectures également massives pour obtenir des résultats agrégés.

Tableau 2-2. Les différences entre OLAP et OLTP

OLTP	OLAP
Mises à jour au fil de l'eau	Insertions massives
Recherches sélectives	Agrégation de valeurs sur de grands ensembles de données
Types de données divers	Principalement données chiffrées et dates
Utilisation partagée entre mises à jour et lectures	Utilisation en lecture seule la plupart du temps
Besoins temps réel	Besoins de calculs sur des données volumineuses, souvent sans exigence d'immédiateté

Les limitations d'OLAP

Le modèle OLAP est l'ancêtre du Big Data. Il a été conçu dans un contexte où le hardware restait cher et où les volumes étaient raisonnables. Il était logique de concentrer ses fonctionnalités sur une seule machine, ou un groupe limité de machines, et de construire un système intégré qui s'occupe du stockage et de l'interrogation des données à travers un modèle organisé en cubes.

Cette conception a pourtant un certain nombre de défauts quand on la ramène aux besoins d'aujourd'hui. Premièrement, elle suppose une modélisation des données préliminaire. Il faut bâtir un modèle en étoile, donc structurer ses données sous forme de table de fait et de table de dimensions. Créer un modèle préalable implique qu'on va structurer les données par rapport à des besoins prédéfinis. Si de nouveaux besoins émergent après la création du système, peut-être que la structure créée ne permettra pas d'y répondre.

Cette structuration suppose en plus qu'on doive utiliser des outils d'importation et de transformation des données (des outils d'ETL) pour préparer la donnée à l'analyse. Cela implique également qu'on ne garde pas forcément tout l'historique des dimensions par exemple, et qu'il faudra au besoin des stratégies de gestion des dimensions changeantes *(slowly changing dimensions)*.

Comme on établit des relations entre une table de fait et des tables de dimensions, des liens subsistent entre les tables, ce qui rend leur distribution plus difficile. De plus, ce modèle fait et dimensions s'applique à une analyse multidimensionnelle des données, mais pas forcément à une recherche de type data mining ou machine learning. On est contraint à certains types de processus.

OLAP *vs* Big Data

Pour répondre à ces limitations, ce qu'on appelle le Big Data présente une conception différente de l'analytique. Une fois de plus, la donnée n'est pas très structurée, en tout cas il n'y a pas dans un système Big Data de schéma explicite de la donnée. L'objectif est de stocker l'intégralité des données, sans choix, sans transformation, sans remodélisation. On stocke la donnée brute, parce qu'on ne sait pas à l'avance ce qu'on va vouloir en faire, quelles sont les informations qu'on va vouloir en tirer. C'est fondamentalement le paradigme Hadoop qui nous a permis de penser de cette manière. Si on y réfléchit, Hadoop est un système qui permet de distribuer des fonctions de traitement sur un cluster de machines qui hébergent également des données.

Dans ce chapitre, nous explorons quelques différences entre le monde SQL et le monde NoSQL. Nous avons parlé de schéma explicite et du langage déclaratif qu'est SQL. Pour que ce système fonctionne, il faut prédéfinir une structure qui soit adaptée aux requêtes à travers ce langage normalisé qu'est SQL, qui travaille sur des ensembles de données structurées de façon précise. Inversement, un langage de programmation de moins haut niveau comme Java offre des usages beaucoup plus généraux, qui ne supposent pas une structuration prédéfinie de la matière avec laquelle il va travailler. Au lieu de définir une requête avec un langage de haut niveau adapté à une certaine structure de données, Hadoop est un framework où l'on développe des fonctions en Java (ou dans un autre langage du même type) qui vont donc s'adapter à la structure des données à disposition, et qui permettent d'envoyer ces fonctions directement plus proches de ces données pour effectuer un traitement. On envoie simplement des fonctions sur un cluster de machines.

Sur Hadoop, il est donc inutile de structurer la donnée. On peut très bien se contenter de fichiers plats, de type CSV par exemple, ou de tout type de structure de base de données ou de sérialisation

comme JSON ou Thrift. Il suffit de stocker au fil de l'eau les données telles qu'elles arrivent, dans toute leur richesse, et ensuite de les traiter au besoin à l'aide de fonctions, et donc de répondre à toutes les demandes futures quelles qu'elles soient.

La conception de Nathan Marz

L'un des acteurs influents du Big Data s'appelle Nathan Marz. Travaillant dans une entreprise rachetée par Twitter en 2011, il a créé une unité au sein de Twitter chargée de développer un environnement de traitement distribué, avant de quitter l'entreprise en 2013 pour fonder sa propre société. C'est le créateur d'Apache Storm, l'une des principales applications de traitement distribué en temps réel. Nous en reparlerons.

Il a mené une réflexion sur le Big Data qui a débouché sur un livre en fin de rédaction chez l'éditeur Manning (http://manning.com/marz/) et sur un concept d'architecture qu'il a appelé l'architecture Lambda. Nous y reviendrons également. Nous nous contenterons ici de parler des principes liés au stockage des données. Un environnement Big Data suivant l'architecture Lambda accueille des données immutables, selon un mode d'ajout uniquement (*append-only*). C'est l'un des fondements du système : la donnée est ajoutée dans toute sa richesse, et elle est conservée à tout jamais. Dans ce cas, la conception de la donnée ressemble à celle d'un journal (un log). C'est le seul moyen de pouvoir tracer un historique complet des données. L'absence de possibilité de mise à jour garantit également qu'il y a aucun risque de perte de données. Tous ceux qui ont déjà utilisé un moteur de base de données relationnelle ont vécu ces deux expériences : un jour, ils ont regretté de ne pas avoir structuré une table avec un historique et d'avoir perdu cette dimension au moment où on leur demandait des informations de ce type ; un autre jour, ils ont perdu des données à cause d'une mauvaise manipulation ou d'un bogue dans le programme client. Le principe de l'architecture Lambda est de ne plus jamais toucher aux données après leur insertion. Les outils de traitements batch de type Hadoop vont permettre de traiter ces grands volumes de données pour produire des résultats sous forme de vues matérialisées et donc de restructurer une copie de ces données brutes pour répondre aux besoins. Admettons qu'il y ait un bogue dans le programme de traitement qui génère ces vues matérialisées. Même si on détecte le bogue six mois après la mise en production, on sera capable de corriger le problème en supprimant les vues matérialisées, en corrigeant le bogue, et en relançant le traitement batch.

La structure des données n'est pas vraiment importante. Comme on envoie des fonctions pour traiter les données, elle peut s'adapter à tous types de format, du fichier plat au moteur de base de données. Donc on sépare vraiment ici le concept de moteur de traitement du concept de stockage. Il vaut mieux d'ailleurs stocker les données dans le format le plus universel possible, par exemple du JSON ou du CSV, de façon à être capable de le lire à partir de n'importe quel langage et dans un futur même lointain, à un moment où un éventuel format propriétaire aurait été déprécié. La structure propre à l'analyse par les outils de lecture et d'interrogation sera implémentée dans les vues matérialisées, qui peuvent donc être facilement recréées.

Décisionnel et NoSQL

Les environnements de traitement distribué de types batch, ou maintenant temps réel, vont peut-être remplacer à terme les outils OLAP du marché. En tout cas, on assiste pour l'instant à une convergence des deux approches, surtout grâce à la popularité de l'approche Big Data. Les éditeurs

de solutions décisionnelles ont tous intégré Hadoop ou un algorithme MapReduce distribué. Nous avons parlé de Microsoft qui supporte maintenant Hadoop avec une plate-forme nommée HDInsight, mais la distribution NoSQL d'Oracle peut également être utilisée avec Hadoop, avec un peu de bricolage (voir http://www.oracle.com/technetwork/articles/bigdata/nosql-hadoop-1654158.html). Un autre exemple est Vertica, un éditeur de solutions décisionnelles qui a été l'un des premiers à développer un connecteur Hadoop (http://www.vertica.com/the-analytics-platform/native-bi-etl-and-hadoop-mapreduce-integration/). Citons également Pentaho, la solution décisionnelle libre (http://www.pentahobigdata.com/ecosystem/platforms/hadoop) qui s'intègre totalement avec Hadoop, offrant un environnement graphique de création de travaux distribués. D'autres sociétés se spécialisent dans la mise à disposition d'environnements décisionnels basés sur Hadoop, telles que Datameer (http://www.datameer.com/), ou développent des interfaces pour Hadoop destinées au décisionnel. L'analytique est donc un des cas d'utilisation majeur de MapReduce.

3

Les choix techniques du NoSQL

Dans les deux chapitres précédents, nous avons présenté le mouvement NoSQL sur un plan historique et par opposition au modèle relationnel. Ces analyses sont intéressantes, mais on peut aussi adopter un troisième point de vue, essentiellement technique. Car si les conceptions ou les idéologies qui gravitent autour des technologies permettent de mieux les comprendre, ce qui compte en définitive, ce sont les technologies elles-mêmes.

Ici, nous allons donc détailler les caractéristiques techniques du NoSQL. On peut les regrouper en deux grandes tendances, qui correspondent à des besoins et à des utilisations spécifiques. Certains moteurs sont ainsi principalement utilisés pour répondre à des problématiques d'interrogation de données très volumineuses (Big Data), d'autres pour assurer le traitement le plus rapide possible des données moins volumineuses mais requérant un flux optimisé, ce qui relève du domaine des bases de données en mémoire.

Nous exposerons dans ce chapitre les choix techniques communs, ainsi que les spécificités des implémentations tendant vers une utilisation Big Data ou vers la performance. Sachez en tout cas que si vous choisissez une solution NoSQL pour vos besoins, vous vous orienterez vers des outils très différents selon que vous êtes dans un cas d'utilisation ou dans l'autre.

L'interface avec le code client

Nous l'avons vu, le langage SQL impose une façon de penser déclarative, ensembliste et parfois fonctionnelle qui n'est pas forcément naturelle pour les développeurs ayant été formés aux

langages impératifs. Les problématiques de défaut d'impédance obligent en plus les développeurs à travailler à travers des chaînes de caractères, ce qui n'est pas très pratique. Les ORM règlent ce problème mais montrent leurs limites lorsqu'on veut optimiser les performances de l'accès aux données. Le mouvement NoSQL est né pour plusieurs raisons, mais un des gains du point de vue du développement est une meilleure intégration du code serveur dans le code client. C'est une qualité des bases NoSQL : elles s'adaptent au développeur et à ses contraintes, au lieu de lui en inventer d'autres. Elles offrent souvent, par exemple, des structures de données proches de celles qui sont déjà utilisées par les développeurs pour l'échange de données ou la sérialisation des objets, comme le JSON. Les pilotes sont implémentés comme bibliothèques des langages de développement les plus utilisés, et il est très facile d'appeler les fonctions des bases NoSQL à travers des bibliothèques d'objets intégrées dans ces langages. Les exemples les plus flagrants de cette facilité sont les entrepôts de paires clé-valeur en mémoire, comme Redis, qui permettent de manipuler des structures qui sont semblables aux structures de données natives des programmes clients (tableaux, tables de hachage). Prenons un exemple très simple de ceci avec Redis. Le code suivant est très court :

```
#!/usr/bin/python

import redis

r = redis.StrictRedis(host='localhost', port=6379, db=2)

r.hmset('contact:ajar', dict(
                prenom='Emile',
                nom='Ajar',
                Profession='Ecrivain'
        )
)

dic = r.hgetall('contact:ajar')
print dic
```

Après avoir ouvert une connexion au serveur Redis et choisi la base de données 2 (en Redis, les bases de données sont indiquées simplement par leur numéro), nous utilisons la méthode `hmset` du client Python, qui correspond à la commande `HMSET` de Redis. Redis est un moteur de paires clé-valeur en mémoire qui offre plusieurs structures de données comme des strings, des sets (ensembles), des tables de hachage, etc. `HMSET` permet de stocker une table de hachage en tant que valeur de la paire. Ici, notre clé est la chaîne `'contact:ajar'`. Nous stockons directement dans cette clé un dictionnaire Python, équivalent d'une table de hachage. Le pilote Redis pour Python s'occupera de faire la traduction pour nous.

Ensuite, nous récupérons notre donnée à l'aide de la méthode `hgetall`, qui extrait une table de hachage stockée dans Redis à partir de la clé fournie en paramètre. Grâce au support des types de données de Python, cette méthode retourne directement un dictionnaire. La dernière commande, le `print` de ce dictionnaire, donne le résultat suivant :

```
{'nom': 'Ajar', 'Profession': 'Ecrivain', 'prenom': 'Emile'}
```

Les fonctionnalités serveur

Alors que dans le monde relationnel, l'utilisation de curseurs ou d'autres types de méthodes procédurales pour manipuler les données de la base est dans la grande majorité des cas une approche à éviter, et qui pose de grands problèmes de performances, les bases NoSQL ont pris la direction inverse. Il n'y a en général aucune complexité déléguée vers le serveur. Le serveur NoSQL retourne un document (une paire clé-valeur) ou une collection de documents, et ces derniers seront ensuite ouverts et parsés par le code client. Les moteurs NoSQL sont souvent utilisés comme de simples entrepôts de données. Le comportement au niveau du serveur n'a pas la richesse du modèle relationnel, où il est possible de créer des contraintes, des déclencheurs, des fonctions, des procédures stockées, bref tout un comportement qui peut parfois les transformer en une couche applicative (un *middle-tier*) à part entière. C'est en tout cas l'état des lieux à l'heure actuelle. Certainement, au fur et à mesure que les moteurs NoSQL vont évoluer, les comportements côté serveur vont s'enrichir. Le paradigme classique pour offrir des fonctionnalités de traitement niveau du serveur, c'est encore une fois MapReduce. Dans CouchDB et Couchbase Server, on peut créer des vues dans des documents de design, qui consiste en deux vues JavaScript (un autre langage peut être choisi) : une vue map qui traite chaque document de la base de données, et une vue reduce facultative qui offre des possibilités d'agrégation. L'intérêt de ce mécanisme est de matérialiser la vue sous forme d'index B-Tree, ce qui offre d'excellentes performances ensuite en interrogation.

Il s'agit donc toujours d'effectuer ce type de traitement : générer des vues pour interroger les données est souvent calculer des agrégats. Riak offre également un mécanisme de MapReduce intégré où des fonctions peuvent être écrites en JavaScript ou en Erlang. Comme Riak est un système distribué décentralisé où n'importe quel nœud peut agir en tant que moteur de requête, une demande MapReduce consiste à envoyer les fonctions map et reduce à l'un des nœuds, qui devient le coordinateur du traitement. Riak étant un moteur clé-valeur, l'accès aux données se fait principalement par la clé, ce qui est assez pauvre pour des besoins traditionnels de requêtage. Vous pouvez maintenant créer des index secondaires pour effectuer des recherches si la partie valeur est composée de documents JSON et, pour tout autre type de traitement, vous pouvez donc envoyer des fonctions MapReduce. La différence avec une requête traditionnelle de SGBDR, c'est que nous ne sommes plus ici dans un comportement de requête en temps réel, mais plus dans une opération de traitements batch avec un temps de traitement qui peut être relativement long selon le volume des données et la complexité du traitement. Il s'agit en quelque sorte de l'ajout d'une fonctionnalité analytique à un moteur opérationnel.

Nous serons amenés à le répéter dans ce livre : avant d'utiliser réellement en production une technologie, renseignez-vous sur la façon dont elle est implémentée dans un moteur particulier. Riak est un système opérationnel dont l'objectif principal est d'offrir de bons temps de traitement sur des données volumineuses avec un dimensionnement élastique. Ses fonctionnalités MapReduce ne doivent pas être considérées comme un outil de premier plan, mais plus comme une possibilité à utiliser en dernier ressort pour couvrir un besoin ponctuel. Le traitement MapReduce en Riak est lent et coûteux pour le cluster, principalement en JavaScript. De plus, vous n'avez pas une garantie absolue de traiter toutes les données, dans le sens où l'algorithme de MapReduce ne prend pas en compte la réplication. Les données vont être cherchées une fois sur un nœud du cluster qui est supposé les avoir, mais si ce nœud ne comporte pas les données ou la dernière version des données, MapReduce n'ira pas chercher ces données ailleurs. Autre chose à prendre

en considération : Riak ne supprime pas physiquement directement ces données mais place des marqueurs de suppression appelés tombstones, pendant trois secondes par défaut. Le traitement MapReduce va s'opérer également sur ces tombstones, et c'est à vous de les filtrer dans le code de votre traitement MapReduce en vérifiant une information dans les métadonnées des objets récupérés. Soyez donc toujours attentif aux contraintes d'implémentation.

Prenons un dernier exemple, celui de MongoDB. C'est un moteur qui a une richesse de requêtage assez rare dans le monde NoSQL, et vous permet d'exprimer des critères de filtrage complexe. Pour un traitement côté serveur, vous pouvez exécuter du code JavaScript en utilisant une fonction nommée `eval()`, ce qui s'avère pratique pour des petites opérations à effectuer sur des collections.

Ainsi, vous pouvez définir une fonction JavaScript, l'exécuter avec un appel à `db.eval()` et récupérer ce résultat dans votre code client. Voici un exemple simple, tiré de la documentation de MongoDB, qui n'est en soi pas très utile car il fait double emploi avec une fonction d'agrégation déjà existante, le `count()` : elle retourne simplement le nombre de documents dans une collection.

```
db.eval( function(){return db[collection].find({},{_id:ObjId()}).length();} );
```

Pour calculer des agrégats, vous pouvez soit utiliser une fonctionnalité MapReduce, soit un framework plus récent nommé Aggregation Framework. Le traitement MapReduce sur MongoDB n'est pas particulièrement rapide, et les gens qui avaient adopté MongoDB pour sa richesse fonctionnelle souffraient ensuite de l'impossibilité d'extraire des données agrégées dans un temps de réponse raisonnable. L'interpréteur MapReduce est JavaScript, d'abord SpiderMonkey, puis à partir de la version 2.4, V8, le moteur JavaScript beaucoup plus rapide de Google. Cela améliore un peu les choses en termes de temps de traitement, mais on peut vraiment mieux faire. MongoDB a donc développé l'Aggregation Framework en C++ et l'a intégré dans le moteur de requête de MongoDB. Nous en verrons un exemple dans le chapitre 8 dédié à MongoDB.

Les protocoles d'accès aux données

Les moteurs NoSQL implémentent différentes méthodes d'accès aux données. Vous avez déjà compris que, quels que soient ces protocoles, des bibliothèques existent pour tous les grands langages clients afin de les rendre plus faciles d'utilisation. Voyons toutefois les interfaces les plus communes entre le client et le serveur.

Interfaces natives

Des serveurs comme MongoDB utilisent un protocole natif pour communiquer avec le serveur. Il s'agit d'une communication en mode question-réponse, basée sur un socket ouvert avec le serveur, sur le port TCP 27017 par défaut. Des messages sont envoyés, composés d'un en-tête et d'un corps de message comprenant des structures assez simples. Par exemple, la structure d'un message d'insertion (qui est identifié par un `opcode` dans l'en-tête qui indique quelle est l'opération demandée) est la suivante :

```
struct {
   MsgHeader header; // en-tête de message contenant l'opcode et d'autres informations
    int32 ZERO; // 0 - réservé pour une utilisation future
    cstring fullCollectionName; // le nom de la collection impliquée
```

```
    BSON[] documents; // un ou plusieurs documents à insérer dans la collection
}
```

Les documents envoyés sont en format BSON, qui est simplement un JSON «binarisé» un peu plus compact (voir le chapitre 4 sur les structures de données).

Il serait donc facile de développer un pilote pour MongoDB. Cela n'est pas vraiment nécessaire car MongoDB Inc., la société qui développe MongoDB, s'est appliqué à réaliser des pilotes *(drivers)* pour la quasi-totalité des langages clients importants, du C++ à Erlang, et liste dans sa documentation les pilotes développés par la communauté (http://wiki.mongodb.org/display/DOCS/Drivers). Ainsi, développer en MongoDB consiste à utiliser des méthodes d'un objet `collection` pour insérer, supprimer, requêter les documents, et ainsi de suite. Quelques exemples de ces appels figurent dans le chapitre 8 de cet ouvrage.

Protocol Buffers

Certains moteurs utilisent un protocole créé et largement utilisé en interne par Google, nommé Protocol Buffers, ou protobuf. Il s'agit d'une définition de format d'échange de données standard qui peut servir à tout type de donnée structurée. Contrairement à MapReduce, protobuf est librement accessible et téléchargeable à l'adresse suivante : http://code.google.com/p/protobuf/. Les moteurs qui implémentent protobuf sont Riak (http://docs.basho.com/riak/1.2.0/references/apis/protocol-buffers/) et HBase (mais pour ses communications internes, interservices uniquement, voir cet article de blog pour les détails techniques : http://blog.zahoor.in/2012/08/protocol-buffers-in-hbase/).

L'utilisation de protobuf est simple : vous définissez une structure de message dans un fichier `.proto`, puis vous utilisez le code fourni par Google pour générer du code de manipulation du message. Le code fourni par Google ne supporte que trois langages : C+, Java et Python. Pour d'autres langages, vous devrez chercher des implémentations réalisées par la communauté, ou faire le travail vous-même en vous inspirant des sources du code de Google. Utilisons notre exemple d'entrée de blog.

Normalement, nous devrions télécharger et compiler le compilateur protobuf disponible à l'adresse Google mentionnée précédemment. Sur Ubuntu, nous n'avons pas besoin de le faire, protobuf est disponible sous forme de paquet, probablement parce que protobuf est utilisé par Canonical pour la fonctionnalité Ubuntu One, son service de stockage sur le cloud. Nous installons donc les paquets nécessaires:

```
sudo apt-get install protobuf-compiler python-protobuf
```

Puis nous créons un fichier `.proto` contenant la définition de notre structure de données :

```
package passerelles;

message Article {
  required int32 id = 1;
  required string titre = 2;

  message Auteur {
    required string prenom = 1;
```

```
    required string nom = 2;
    required string email = 3;
  }

  repeated Auteur auteur = 3;
}

message Articles {
  repeated Article article = 1;
}
```

Ce document illustre quelques-unes des possibilités de protobuf. La structure est hiérarchique : un message peut contenir d'autres messages, et ces messages peuvent être indiqués comme étant présents plusieurs fois dans le message contenant, avec le mot-clé `repeated`. Chaque message est composé de types de données scalaires simples (la liste des types supportés est disponible à l'adresse suivante : https://developers.google.com/protocol-buffers/docs/proto?hl=fr#scalar) ou d'autres messages. Chaque élément peut être déclaré comme obligatoire (`required`) ou optionnel (`optional`), et une valeur par défaut peut être spécifiée pour les éléments optionnels, qui sera utilisée lorsque le code parse un message protobuf où l'élément n'est pas présent. Finalement, dans la définition du message, un numéro indiquant la séquence des éléments est obligatoire.

Énumérations

Nous ne les avons pas utilisées dans notre exemple, mais protobuf supporte également des énumérations (des listes de valeurs) comme type de donnée pour des éléments du message. Ces énumérations se définissent directement dans le message, en tant que structure `enum`.

Ce fichier écrit et créé, il nous reste à générer le code de gestion de cette structure. Pour ce faire, nous utilisons le compilateur protobuf `protoc`, que nous appelons ici dans le répertoire où nous avons sauvegardé notre fichier :

```
protoc -I=./ --python_out=./ ./passerelles.proto
```

Le compilateur prend un répertoire source (`I` pour *input*), un type de destination (nous indiquons `python_out` pour générer du code Python dans le répertoire courant) et un fichier `.proto` d'entrée. Le résultat est un fichier Python nommé `<nom du package>_pb2.py`, ici `passerelles_pb2.py`. Le fichier Python généré contient des descripteurs de format et définit des classes pour créer des messages. Nous en reproduisons ici la signature uniquement :

```
class Article(message.Message):
  __metaclass__ = reflection.GeneratedProtocolMessageType

  class Auteur(message.Message):
    __metaclass__ = reflection.GeneratedProtocolMessageType
    DESCRIPTOR = _ARTICLE_AUTEUR

  DESCRIPTOR = _ARTICLE
```

```
class Articles(message.Message):
  __metaclass__ = reflection.GeneratedProtocolMessageType
  DESCRIPTOR = _ARTICLES
```

Les classes définies ici utilisent le concept de métaclasse en Python, qui permet d'injecter par réflexion des fonctionnalités à une classe existante. Ceci sort un peu du cadre de ce livre, considérez simplement que le code nécessaire à la gestion du message est injecté dans la classe que vous voyez ici. Utilisons le module généré dans un code d'exemple Python :

```
#! /usr/bin/python
# -*- coding: utf-8 -*-

import passerelles_pb2

article = passerelles_pb2.Article()
article.id = 1
article.titre = unicode(«pourquoi les éléphants ont-ils de grandes oreilles?»,
'utf-8')
auteur = article.auteur.add()
auteur.prenom = «Annie»
auteur.nom = «Brizard»
auteur.email = «annie.brizard@cocomail.com»

f = open("exemple.msg", "wb")
f.write(article.SerializeToString())
f.close()
```

Ici, nous importons le module `passerelles_pb2` généré par `protoc` et nous créons une instance de notre article, que nous alimentons avec des données. Nous convertissons le titre en utf-8 pour que les caractères accentués soient correctement encodés par protobuf. Finalement, nous utilisons la méthode `SerializeToString()` pour écrire le résultat dans un fichier. Le fichier `exemple.msg` contient une chaîne compacte du contenu du message avec seulement quelques caractères de contrôle pour séparer les éléments.

Cet exemple montre à quel point protobuf est simple à utiliser. On peut s'en servir pour échanger des informations en RPC entre un client et un serveur, comme c'est le cas dans son utilisation avec Riak. Une implémentation en RPC pour protobuf utilisant des sockets est disponible en Python à cette adresse : http://code.google.com/p/protobuf-socket-rpc/.

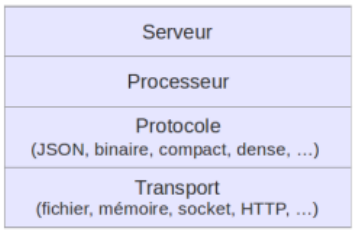
Le fait qu'un message protobuf sérialisé soit assez compact pourrait aussi inciter à l'utiliser comme format de stockage. Ce n'est pas un choix fait par les grands moteurs NoSQL, mais certains l'ont jugé intéressant, notamment les développeurs de Twitter qui l'utilisent pour stocker des milliards de tweets. Une présentation sur le sujet est disponible à l'adresse suivante : http://fr.slideshare.net/kevinweil/protocol-buffers-and-hadoop-at-twitter.

Thrift

Thrift est un autre protocole, en réalité un framework initialement développé et utilisé en interne par Facebook, et donné en 2007 à la fondation Apache. Il est utilisé comme interface, soit

directement, soit de bas niveau, par certains moteurs NoSQL. Tout comme protobuf, il permet de définir une structure de message. On y ajoute une interface de service qui permet de définir les opérations supportées sur cette structure, et un compilateur génère l'implémentation dans le langage souhaité. Thrift est encore un peu plus que cela : il incorpore tous les niveaux nécessaires à la création de services. Les différents niveaux de Thrift sont illustrés sur la figure 3-1.

Figure 3-1
La pile Thrift



Une couche serveur permet de définir différents types de serveurs, qu'ils soient *single-thread* ou *multi-thread*. La couche processeur est le code généré par Thrift pour effectuer les traitements des messages définis. La couche protocole permet d'empaqueter les données dans des formats comme JSON, compact, dense, binaire, etc. Enfin, la couche transport met à disposition le code pour échanger les données, par exemple en fichiers disque ou via sockets, HTTP, etc. Il s'agit donc d'une bibliothèque complète de développement de services.

Thrift supporte nativement beaucoup plus de langages que protobuf, en voici une liste non exhaustive : C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk et Ocaml.

Voyons comment fonctionne Thrift en développant un petit exemple. Contrairement à protobuf, Thrift n'est pas disponible sous forme de paquets binaires pour Ubuntu. Nous devons donc le télécharger et le compiler. Installons au préalable les paquets nécessaires à la compilation :

```
sudo apt-get install libboost-dev libboost-test-dev libboost-program-options-dev
libevent-dev automake libtool flex bison pkg-config g++ libssl-dev
```

Ensuite, téléchargeons la version actuelle de Thrift, disponible sur son site http://thrift.apache.org/download/, et compilons-la :

```
wget https://dist.apache.org/repos/dist/release/thrift/0.9.0/thrift-0.9.0.tar.gz
tar xvzf thrift-0.9.0.tar.gz
cd xvzf thrift-0.9.0
./configure
```

L'appel de `configure` va tester la présence de langages sur notre système. Ici, ce qui nous intéresse est l'utilisation avec Python. Celui-ci étant présent sur notre système, la configuration affiche, entre autres, cette information :

```
Python Library:
   Using Python ..............: /usr/bin/python
```

Si vous voulez utiliser Thrift avec un langage, veillez donc à ce qu'il soit préalablement installé, et soyez attentif à la présence de cette information vers la fin de l'output de `configure`. De même, si vous êtes sur Ubuntu, la compilation de Thrift pour votre langage nécessite l'installation de certains paquets, listés sur cette page d'aide : http://thrift.apache.org/docs/install/ubuntu/. De notre côté, afin d'éviter d'installer les dépendances nécessaires pour le langage Ruby qui était installé sur notre système, nous avons utilisé l'option `--without-ruby` de `configure` :

```
./configure -without-ruby
```

La liste des options de `configure` est disponible avec :

```
./configure --help
```

Ensuite, nous compilons, exécutons les tests et installons Thrift avec `make` :

```
make
make check
sudo make install
```

Nous testons si Thrift est correctement installé :

```
thrift -version
```

ce qui retourne :

```
Thrift version 0.9.0
```

Nous installons alors le module Python pour utiliser Thrift :

```
sudo pip install thrift
```

Tout comme pour protobuf, nous créons un fichier de définition de données. Essayons ici de le créer au plus proche de ce que nous avons fait pour protobuf.

```
namespace py passerelles

struct Auteur {
    1:required string prenom,
    2:required string nom,
    3:required string email
}

struct Article {
  1:required i32 id,
  2:required string titre,
  3:required Auteur auteur
}

service Articles {
    void ping();
```

```
    bool addArticle(1:Article article);
    Article getArticle(1:i32 id);
}
```

Nous déclarons d'abord un espace de nom optionnel. Thrift nous permet de créer des espaces de noms différents pour chaque langage, c'est la raison pour laquelle nous indiquons ici `py` pour notre compilation pour Python. Nous créons ensuite une structure pour l'auteur, et une structure pour l'article. Le fonctionnement est assez semblable à protobuf, les éléments sont numérotés en tête au lieu de l'être à la fin, et le mot-clé `struct` est utilisé pour définir une structure. Des `enum` peuvent aussi être utilisés et une `struct` peut être employée dans une autre. Finalement, nous définissons un service, qui est une interface comprenant la signature des méthodes que nous voulons voir créées dans le code généré. Nous sauvegardons cette définition dans un fichier que nous nommons `article.thrift`, puis nous générons le code Python avec le compilateur Thrift :

```
thrift -r --gen py article.thrift
```

L'option `--gen py` nous permet d'indiquer que la cible est du code Python. Le résultat est généré dans `./gen-py/passerelles`, selon l'espace de nom que nous avons défini. Le code généré est un peu plus complexe que pour protobuf, mais basiquement, nous nous retrouvons avec un fichier `Articles.py` qui contient la définition du service, et un script nommé `Articles-remote` qui permet d'utiliser en ligne de commande les fonctionnalités de transport de Thrift.

Thrift était le protocole bas niveau de communication avec Cassandra jusqu'à sa version 2 sortie en 2014. Cassandra définissait une interface sous forme de fichier Thrift qui donnait toutes les structures et fonctions pour communiquer avec le serveur. Cette interface est toujours disponible et s'appelle le mode d'accès par l'API. À partir de la version 2, l'accès API est déprécié et le protocole officiel s'appelle CQL, pour Cassandra Query Language. Il s'agit d'une implémentation d'un langage déclaratif très proche du SQL. On pourrait donc considérer que Cassandra n'est plus un moteur NoSQL. Nous reparlerons également de cette décision de la communauté de développement de Cassandra de changer radicalement de mode d'accès. Cela a en tout cas amélioré les choses en termes de rapidité, parce que l'accès CQL est natif et binaire, et les benchmarks ont montré de nets gains de performance.

D'autres moteurs utilisent Thrift, notamment HBase (http://wiki.apache.org/hadoop/Hbase/ThriftApi). Il représente une bonne solution pour accéder à HBase à partir de clients non Java, car l'offre de pilotes « natifs » pour des langages autres que Java n'est pas encore très étendue en ce qui concerne HBase. Par ailleurs, l'outil ElasticSearch (http://www.elasticsearch.org/guide/reference/modules/thrift.html), proche du monde NoSQL, présente une interface Thrift.

Thrift est également l'interface pour Apache Storm, le framework de traitement de flux en temps réel distribué à la popularité croissante. Thrift est utilisé par Storm pour la communication des données entre ses composants sur un cluster, et l'API de bas niveau à Nimbus, la machine maître de Storm, est en Thrift.

HiveServer et HiveServer2 sont des services optionnels intégrés à Apache Hive, une infrastructure très utilisée, basée sur Hadoop, pour permettre à des clients d'utiliser les fonctionnalités de requêtage de Hive ; ils utilisent tous deux une interface Thrift. C'est également le cas du serveur

JDBC d'Apache Spark, un successeur très intéressant et beaucoup plus complet de Hive, qui offre un framework complet de traitement distribué.

Apache Avro

Il existe également un autre système de sérialisation de la fondation Apache, nommé Avro (http://avro.apache.org/). Il est plus récent et offre une certaine souplesse par rapport à protobuf et Thrift, surtout parce qu'il n'est pas nécessaire de le compiler au préalable comme c'est le cas pour les deux protocoles précédents. Le schéma est défini en JSON avec la donnée, ce qui permet de traiter les structures en Avro dynamiquement. De plus, les champs n'ont pas besoin d'être numérotés comme dans protobuf et Thrift. Une tentative a été faite de développer une interface Avro pour Cassandra, qui devait théoriquement remplacer Thrift, mais principalement par manque de popularité, ce projet n'a pas été suivi. Avro est peut-être une solution d'avenir, elle commence à bien s'intégrer dans l'écosystème Hadoop (voir http://blog.cloudera.com/blog/2011/07/avro-data-interop/) et fait partie des développements gérés par Doug Cutting (il en a écrit les spécifications, voir http://avro.apache.org/credits.html). C'est donc un développement à surveiller. Deux modules existent pour Python : le premier, nommé simplement Avro, est relativement lent, c'est la raison pour laquelle un second module compilé en C est aussi disponible, fastavro (http://pypi.python.org/pypi/fastavro/).

REST

Pour l'instant, nous avons exploré des protocoles plutôt classiques, de type RPC *(Remote Procedure Call)*, qui lient un client et un serveur avec des interfaces bien définies et assurent en général le maintien d'une connexion TCP. Une autre vision de la communication entre machines est possible, sous la forme d'architecture orientée services ou d'architecture orientée ressources, dans le cas de REST. REST *(Representational State Transfer)* est une architecture spécifique à l'appel de ressources disponibles dans un environnement hétérogène et distribué – le Web, en un mot. Elle a été définie par Roy Fielding, l'un des principaux auteurs de la spécification HTTP et un contributeur au développement du serveur HTTP Apache.

REST se base sur le protocole de base du Web, le HTTP. L'idée est simple et c'est ce qui la rend intéressante. Le protocole HTTP est basé sur l'échange de documents et il comporte des verbes, ou méthodes, qui permettent d'indiquer un type d'action voulue à un serveur HTTP, en général un serveur web. Pourquoi ne pas simplement utiliser ces verbes pour communiquer avec des ressources et des services distants ? Tous les langages de programmation ont déjà des bibliothèques qui permettent de communiquer de cette manière. De plus, une demande REST est sans état, toute la demande est contenue dans son appel, et tout le résultat est renvoyé en résultat. Il est inutile, à la base, de maintenir une connexion. C'est simple et élégant. Cela permet aussi aux serveurs de simplifier leurs interfaces avec le client : nul besoin de maintenir une connexion ouverte, de conserver une vue cohérente des données durant la vie de cette connexion, etc. Plusieurs moteurs NoSQL ou apparentés (nous pensons ici à ElasticSearch) implémentent une interface REST. Le plus évident est CouchDB, mais des moteurs comme :

- Riak – http://docs.basho.com/riak/latest/references/apis/http/
- HBase – http://wiki.apache.org/hadoop/Hbase/Stargate
- Neo4J – http://docs.neo4j.org/chunked/milestone/rest-api.html

offrent également des interfaces REST.

Un fournisseur de ressources qui respecte les règles du modèle REST est dit *RESTful*. Ces règles sont assez simples : séparation des responsabilités entre client et serveur, aucun maintien d'état entre des demandes, découplage du client et du serveur par utilisation d'interfaces, etc. Les verbes HTTP s'enrichissent au fil des versions des RFC (*Request For Comments*, documents qui définissent les aspects techniques d'Internet) et ne sont pas forcément tous supportés par la majorité des serveurs HTTP. Une interface REST pourrait au besoin en utiliser toute la richesse, mais en pratique, toutes les opérations nécessaires peuvent être réalisées à l'aide de six méthodes.

Tableau 3-1. Les méthodes REST

Méthode	Utilisation
`GET`	Méthode utilisée pour récupérer une représentation d'une ressource, c'est-à-dire tout type de donnée.
`HEAD`	Comme `GET`, méthode qui permet d'envoyer une demande mais elle ne retourne pas le résultat, seulement l'en-tête. Utile pour récupérer des métadonnées.
`PUT`	Méthode permettant de créer une ressource, de stocker une représentation d'une ressource. Elle permet donc l'insertion et la mise à jour. La règle veut qu'un envoi ultérieur de la même ressource (portant le même identifiant) remplace l'ancienne ressource. Dans certains moteurs comme CouchDB, ceci est réalisé par la méthode `PUT` en créant une nouvelle version du document.
`POST`	Méthode qui offre à peu près les mêmes fonctionnalités que `PUT`, mais d'une façon plus souple. On considère que `POST` envoie une requête au serveur, alors que `PUT` envoie une ressource à stocker, cette dernière étant indiquée dans l'`URL` (voir plus loin). `POST` peut aussi être utilisée pour transmettre une demande d'action au fournisseur de la ressource, sans qu'il y ait stockage de données.
`DELETE`	Méthode qui permet simplement de supprimer une ressource.
`OPTIONS`	Méthode qui retourne les ressources disponibles sur un serveur.

L'utilisation de ces méthodes sera illustrée un peu plus loin.

Dans le monde du Web, la méthode `GET` est la plus répandue. Par exemple, pour accéder au site des éditions Eyrolles, un navigateur web envoie une requête HTTP dont les deux premières lignes pourraient être :

```
GET / HTTP/1.1
HOST: www.editions-eyrolles.com
```

Le verbe `GET` demande un document, il indique le chemin souhaité, ici la racine (/). Ensuite, l'en-tête de la requête est constitué d'une liste de paires clé-valeur. Nous avons seulement montré la première, l'indication du serveur à qui envoyer la demande.

Exemple avec une ressource REST : parse.com

Pour illustrer l'utilisation de ressources REST à l'aide d'un langage client, nous allons utiliser un service en ligne qui propose une interface REST. Il en existe beaucoup désormais, le plus évident est Amazon S3 *(Simple Storage Service)* qui offre un système de stockage dans le cloud. Nous avons trouvé un service intéressant qui nous permettra de démontrer simplement un échange de données avec REST : le service parse.com (https://www.parse.com/). Parse est une

société californienne qui propose un système de gestion de données en ligne à destination des développeurs d'applications mobiles. Elle prend en charge la partie serveur de ces applications. Des API sont disponibles pour IOS, Android, Windows 8 et une API REST est également disponible. Nous allons utiliser l'outil curl et du code Python pour communiquer avec le site parse.com. La documentation de l'API REST de parse.com est disponible à l'adresse suivante : https://www.parse.com/docs/rest.

curl

La façon la plus simple de tester un service REST est d'utiliser l'outil en ligne de commande curl. Pour rappel, curl (*Client URL Request Library*, soit bibliothèque de requêtes aux URL des clients en français, http://curl.haxx.se/) est un outil courant du monde Unix qui permet d'interagir avec des URL, en utilisant à peu près tous les protocoles d'Internet, et avec une richesse fonctionnelle qui en fait un vrai couteau suisse du transfert de données sur le Web. Curl n'est pas toujours installé par défaut sur les distributions. Vous pouvez l'obtenir sur Debian ou Ubuntu en installant le paquet `curl` :

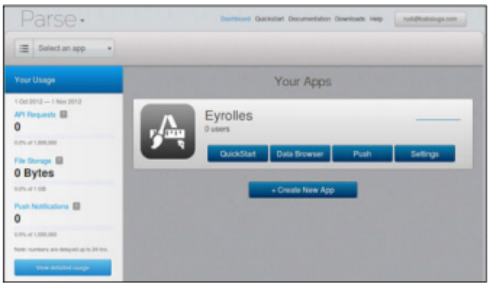
```
sudo apt-get install curl
```

Curl est installé par défaut sur Mac OS X et disponible en binaires pour Windows à l'adresse suivante : http://curl.haxx.se/download.html, qui liste les ports de curl sur tous les systèmes possibles et imaginables.

> **L'avantage de curl**
>
> L'utilisation de curl permet d'étudier très simplement le fonctionnement naturel d'une ressource REST, sans aucune couche d'abstraction amenée par une bibliothèque. En utilisant l'option `-v`, nous pouvons en plus voir avec curl tous les détails de l'envoi et de la réception des données, notamment les en-têtes HTTP, ce qui est très pratique pour déboguer. Pour afficher plus de détails lorsque vous exécutez une commande `curl` comme celles que nous allons voir, remplacez l'option `-X` par `-vX`.

Nous avons ouvert un compte sur parse.com et nous avons créé une application nommée Eyrolles. Nous utilisons le plan basique, gratuit, qui offre un giga-octet de stockage et un million de requêtes et de *push* de données par mois. Sur notre tableau de bord (https://www.parse.com/apps), nous pouvons visualiser les applications disponibles et l'utilisation du service, comme illustré sur la figure 3-2.

Figure 3-2
Application sur parse.com



Chaque application dispose d'un identifiant propre *(Application ID)* et d'une clé client *(Client Key)* qui permet l'authentification à partir du code client.

Une application parse.com stocke les données dans un document composé des paires clé-valeur avec des types de données compatibles JSON (voir le chapitre suivant consacré aux schémas de données dans les bases NoSQL). Ce document est appelé un PFObject et il est contenu dans une classe, ce qui lui confère plus ou moins une fonctionnalité de collection de documents. Tous les appels à l'API REST se font sur le domaine https://api.parse.com. Un exemple d'URL serait : https://api.parse.com/1/classes/Passerelles/CAQr2WT1r0, qui correspond donc à l'adresse d'une ressource, en l'occurrence le document portant l'ObjectId `CAQr2WT1r0`, dans la classe `Passerelles`. L'adresse https://api.parse.com/1/ indique que nous accédons à travers la version 1 de l'API REST. La méthode `POST` nous permet d'ajouter un document. Utilisons donc curl pour ajouter notre article de blog :

```
curl -X POST https://api.parse.com/1/classes/Passerelles \
  -H "X-Parse-Application-Id: Cf5HjUWTzRVYNtLEjP5T4ys74bk5tsVg2ByayAA5" \
  -H "X-Parse-REST-API-Key: xtefrYwFDkXtovRDpo08isxfjhPAGAvQ88fI9Mn" \
  -H «Content-Type: application/json» \
  -d '
{
«prenom»:»Annie»,
«nom»:»Brizard»,
«email»:»annie.brizard@cocomail.com»,
«titre»:»pourquoi les éléphants ont-ils de grandes oreilles?»
}'
```

Ici, nous indiquons à curl que nous effectuons un `POST` sur la ressource https://api.parse.com/1/classes/Passerelles. Nous n'avons pas créé la classe `Passerelles` à l'avance, elle sera automatiquement créée au premier `POST` d'un PFObject dans cette classe. Les paramètres `-H` permettent d'envoyer des paires clé-valeur d'en-tête HTTP. Ici, nous devons fournir l'identifiant de l'application ainsi que la clé client, et indiquer que le `Content-Type` est du JSON, indispensable pour communiquer avec l'API parse.com. Ensuite, nous envoyons le document à l'aide du paramètre `-d` *(data)*. Nous avons dû simplifier notre document, parse.com n'accepte pas les documents JSON complexes, et nous oblige à créer des clés sans accents et signes non alphanumériques.

parse.com nous répond avec un JSON de résultat, indiquant la date de création et l'ObjectId généré :

```
{"createdAt":"2012-11-01T13:13:24.375Z","objectId":"CAQr2WT1r0"}
```

Nous pouvons donc maintenant retrouver notre donnée à l'aide d'un `GET` :

```
curl -X GET https://api.parse.com/1/classes/Passerelles/CAQr2WT1r0 \
  -H "X-Parse-Application-Id: Cf5HjUWTzRVYNtLEjP5T4ys74bk5tsVg2ByayAA5" \
  -H "X-Parse-REST-API-Key: xtefrYwFDkXtovRDpo08isxfjhPAGAvQ88fI9Mn" \
  -H «Content-Type: application/json»
```

qui nous retourne le JSON du document :

```
{«prenom»:»Annie»,»nom»:»Brizard»,»email»:»annie.brizard@cocomail.
com»,»titre»:»pourquoi les éléphants ont-ils de grandes oreilles?»,»createdAt»:»2012-
11-01T13:13:24.375Z»,»updatedAt»:»2012-11-01T13:13:24.375Z»,»objectId»:»CAQr2WT1r0»}
```

Nous pouvons d'ailleurs aussi le constater dans l'explorateur de données du tableau de bord de parse.com (https://www.parse.com/apps/eyrolles/collections), qui nous offre une vision tabulaire du document JSON (figure 3-3).

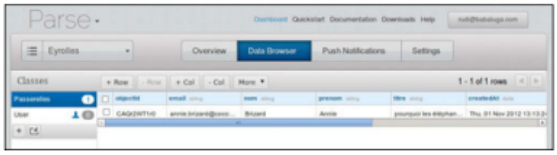


Figure 3-3
Vision du document JSON sur parse.com

Pour modifier un objet, nous utilisons la méthode `PUT` :

```
curl -X PUT https://api.parse.com/1/classes/Passerelles/CAQr2WT1r0 \
  -H "X-Parse-Application-Id: Cf5HjUWTzRVYNtLEjP5T4ys74bk5tsVg2ByayAA5" \
  -H "X-Parse-REST-API-Key: xtefrYwFDkXtovRDpo08isxfjhPAGAvQ88fI9Mn" \
  -H «Content-Type: application/json» \
  -d '{«titre»:»les éléphants ont-ils vraiment bonne mémoire?»}'
```

Le document doit contenir les clés à modifier, le reste du document ne sera pas touché. Nous recevons une notification de mise à jour :

```
{«updatedAt»:»2012-11-01T13:57:14.749Z»}
```

Enfin, nous pouvons supprimer l'objet grâce à la méthode `DELETE` :

```
curl -X DELETE https://api.parse.com/1/classes/Passerelles/CAQr2WT1r0 \
  -H "X-Parse-Application-Id: Cf5HjUWTzRVYNtLEjP5T4ys74bk5tsVg2ByayAA5" \
  -H "X-Parse-REST-API-Key: xtefrYwFDkXtovRDpo08isxfjhPAGAvQ88fI9Mn"
```

L'objet est alors silencieusement supprimé.

Avantages et inconvénients de l'interface REST

REST est un paradigme d'accès aux ressources très élégant et extrêmement simple à mettre en œuvre. Tous les langages de développement comportent des bibliothèques qui permettent d'utiliser le protocole HTTP. En proposant une interface REST, les fournisseurs de ressources et certains moteurs NoSQL offrent une solution facile d'accès aux données. Alors, pourquoi ne pas se simplifier la vie et utiliser une interface REST pour tous vos appels vers les moteurs de bases de données NoSQL qui le supportent ? Malheureusement, parce que HTTP et REST ne sont pas des protocoles conçus pour la rapidité. Si on prend l'exemple de CouchDB, qui offre uniquement une interface d'accès en REST, c'est un moteur riche fonctionnellement mais trop lent pour offrir une solution intéressante pour des applications qui ont besoin de manipuler des volumes honorables de données avec une faible latence. Ce n'est pas dû uniquement à REST et une initiative comme Couchbase (http://www.couchbase.com/), dirigée aussi par Damien Katz, le créateur de CouchDB, propose un développement autour de CouhDB qui résout un certain nombre de problèmes, en offrant, par exemple, un sharding automatique et une intégration de memcached. Techniquement, Couchbase est une combinaison de CouchDB et de Membase, un entrepôt de paires clé-valeur basé sur memcached. L'accès aux données dans Couchbase est effectué via le protocole binaire de memcached, qui offre de bien meilleures performances que REST et HTTP.

En conclusion, selon vos besoins, l'interfaçage en REST avec un dépôt de données NoSQL peut s'avérer être une bonne solution technique, par sa simplicité et son universalité. Cependant, elle montrera vite ses limites si vous avez besoin de bonnes performances.

L'intérêt des protocoles pour la persistance

Les frameworks comme Thrift ou Protobuf incluent le support de toutes les couches nécessaires à la manipulation et à l'échange d'information structurée ou semi-structurée. Même si vous n'avez pas besoin d'échanger des informations entre services, vous pouvez a minima en tirer parti pour leur capacité à exprimer et à persister des structures de données. Pour un projet Big Data par exemple, nous avons déjà évoqué la séparation entre données et traitement, et les critères particuliers de la donnée : immutable, complète, brute, traitée ensuite en batch pour générer des vues matérialisées. Vous pouvez ainsi très bien utiliser Thrift pour structurer la donnée que vous allez manipuler dans un tel système. Cela vous permet d'avoir une donnée structurée qui reste universellement accessible à partir de n'importe quel langage, et dont le format est pérenne.

Ajoutons une chose : que vous utilisiez du CSV, du Thrift, de l'Avro, peu importe, c'est une bonne idée aussi de compresser les données, par exemple avec un algorithme de type Lempel-Ziv ou dérivé. Vous allez pouvoir stocker beaucoup plus de données sur les mêmes nœuds, et vous allez améliorer les performances d'entrées-sorties, en chargeant certes un peu plus les processeurs, mais en général le compromis est bénéfique pour les performances. Comme vous allez lancer des opérations en batch, vous n'êtes de toute façon pas très sensible à ces performances.

Prenons l'exemple d'Hadoop MapReduce. Historiquement, la structure de données de base utilisée par les fonctions mapper et reducer est un objet `Writable` qui représente des paires clé-valeur. `Writable` implémente un protocole de sérialisation simple. Cette implémentation date du développement original de Doug Cutting pour faire fonctionner Hadoop pour Nutch (voir notre petit historique dans le chapitre 1). À partir de la version 0.17 qui date de 2008, vous pouvez utiliser

n'importe quel protocole de sérialisation. Auparavant, la déclaration d'une fonction mapper était la suivante :

```
class MyMapper implements Mapper {
  public void map(WritableComparable key, Writable val,
    OutputCollector out, Reporter reporter) throws IOException {
    // ...
  }
  // ...
}
```

alors qu'elle utilise maintenant (Hadoop 2.6.0) des génériques définis au niveau de la classe Mapper :

```
Class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT> extends Object
```

Et la définition de la méthode attend bien sûr les types définis à la création de l'objet :

```
protected void map(KEYIN key, VALUEIN value, org.apache.hadoop.mapreduce.Mapper.
Context context) throws IOException, InterruptedException
```

Les types génériques

Les types génériques présents dans des langages comme Java ou C# permettent de créer des classes manipulant des données dont le type n'est pas connu à l'avance. Lorsque vous créez une instance de la classe, vous indiquez quel est le type que cette instance va manipuler. Le typage de la classe sera donc effectué à la compilation, puisque le compilateur sait à partir de la déclaration d'instanciation de quel type il s'agit. Vous pouvez ainsi créer des classes réutilisables qui restent type-safe.

Donc vous pouvez utiliser n'importe quelle structure dans un travail MapReduce. De plus, tout le travail pénible a déjà été fait par Twitter dans sa bibliothèque libre elephant-bird (https://github.com/twitter/elephant-bird) qui offre des objets `InputFormat`, `OutputFormat` et `Writable` pour Hadoop en Thrift ou Protobuf avec le support de la compression LZO. C'est une bibliothèque très utilisée par Twitter en interne pour leurs besoins.

Vous trouvez des exemples complets de MapReduce avec Lzo, Thrift et Protobuf dans les sources d'elephant-bird à cette adresse : https://github.com/twitter/elephant-bird/tree/master/examples/src/main/java/com/twitter/elephantbird/examples.

L'architecture distribuée

Comme nous l'avons vu, l'une des raisons qui ont conduit à la naissance du NoSQL est de pouvoir distribuer les données sur plusieurs machines, afin d'assurer la gestion de données de plus en plus volumineuses, ainsi qu'une une montée en charge la plus rapide et indolore possible. L'architecture distribuée est donc au cœur du NoSQL. Il existe deux grandes façons de considérer la distribution des traitements et des données : avec ou sans maître. Les moteurs NoSQL utilisés pour gérer de grands volumes de données ont fait l'un ou l'autre choix, nous allons voir lesquels et comment ils fonctionnent.

La distribution avec maître

La première solution de la distribution ou du partitionnement de données s'appuie sur la présence d'une machine maître, qui conserve la configuration du système, reçoit les requêtes des clients et les redirige vers les machines qui contiennent les données désirées.

La réplication maître-esclave

La réplication n'est pas de la distribution, c'est une copie des données qui sert avant tout à assurer la redondance, donc la protection contre la perte de données. C'est une technologie courante dans le monde relationnel, et à peu près tous les moteurs relationnels importants permettent de mettre en place une réplication maître-esclave. Elle est importante pour répondre à l'exigence de cohérence transactionnelle forte, qui rend difficile les écritures sur plusieurs serveurs, à moins de mettre en place un système de résolution de conflits. On écrit donc sur le maître, qui se charge de répliquer les données sur les esclaves. Ces machines secondaires peuvent, en revanche, être utilisées en lecture, tout dépend du besoin de cohérence de la lecture. En effet, permettre la lecture sur un réplica implique qu'il est possible de lire une donnée obsolète, qui viendrait d'être mise à jour sur le serveur maître, mais pas encore répliquée. Dans le monde NoSQL, ce risque dépend du caractère synchrone ou asynchrone de la mise à jour, comme nous l'avons vu.

Exemple de MongoDB

En ce qui concerne la réplication purement maître-esclave, elle est utilisée dans MongoDB, où elle est traditionnellement alliée au sharding pour partitionner les données et assurer la redondance. Afin de s'assurer de la durabilité de la modification, on peut indiquer une option à l'envoi de la modification au serveur MongoDB, ce que MongoDB Inc. appelle un *write concern*, ou en français prévenance d'écriture. Par défaut, une opération d'écriture sur MongoDB est asynchrone : le code client continue son exécution sans aucune garantie du succès de l'écriture sur le serveur.

La méthode pour définir le write concern a changé depuis l'édition précédente de ce livre. Traditionnellement, on pilote ce comportement à l'aide de la commande `getLastError`, qui détermine une option de sécurité d'écriture. Les options disponibles sont mentionnées dans le tableau 3-2.

Tableau 3-2. Write concern en MongoDB

Option	Description
vide	MongoDB a reçu la modification en mémoire.
`J` (journal)	La donnée est écrite dans le journal (*WAL, Write-Ahead Log*).
`W`	En cas d'ensemble de réplicas, cette option définit le nombre de réplicas où la donnée doit avoir été écrite pour considérer l'écriture réussie.

L'option `w` prend comme paramètre un nombre de réplicas sur lesquels l'écriture doit avoir réussi, par défaut 1. Vous pouvez aussi indiquer la valeur `majority` pour indiquer que l'écriture doit avoir réussi sur une majorité de réplicas.

Voici un exemple de commande JavaScript (à saisir, par exemple, dans l'invite de commandes de MongoDB, voir le chapitre 8 dédié à MongoDB pour plus de précisions) pour activer cette option dans les anciennes versions de MongoDB :

```
db.runCommand( { getLastError: 1, w: "majority" } )
```

Cette option est définie par session, et est en général intégrée dans le pilote du langage que vous utilisez. Par exemple, le paramètre `safe` du pilote Python activait `getLastError` :

```
collection.save({"name": "MongoDB"}, safe=True)
```

Elle peut également être modifiée dans la configuration de l'ensemble de réplicas pour affecter le comportement par défaut de toutes les écritures (effectuées sans indication explicite de `getLastError`), à l'aide de l'objet de configuration `rs.conf()` de l'objet `rs` (*Replica Set*, ensemble de réplicas).

Le code suivant permet de modifier la configuration par défaut, à exécuter dans l'invite JavaScript de MongoDB :

```
cfg = rs.conf()
cfg.settings = {}
cfg.settings.getLastErrorDefaults = {w: "majority", j: true}
rs.reconfig(cfg)
```

À partir de MongoDB 2.6, une mise à jour importante sortie en avril 2014, le write concern est plus logiquement intégré dans le nouveau protocole d'écriture, qui ajoute les commandes `insert`, `update` et `delete`. Ces nouvelles commandes, intégrées dans les pilotes pour les langages clients, comportent un paramètre où le write concern peut être mentionné explicitement. La valeur par défaut reste définie par `getLastError`. Voici pour exemple la syntaxe de la commande `insert` :

```
{
   insert: <collection>,
   documents: [ <document>, <document>, <document>, ... ],
   ordered: <boolean>,
   writeConcern: { <write concern> }
}
```

Voici un exemple de valeurs pour le paramètre :

```
writeConcern: { w: «majority», wtimeout: 5000 }
```

ce qui signifie, un write concern à la majorité, avec un délai d'attente de 5 secondes (5000 millisecondes). L'option `wtimeout` peut être spécifiée globalement avec `getLastError` et existe donc dans ce contexte aussi dans les versions antérieures.

Ces nouvelles commandes permettent également de recevoir un document `writeConcernError` qui contient le code et le message d'une éventuelle erreur de write concern, comme le dépassement du délai d'attente.

Lecture sur un esclave

Vous pouvez aussi configurer MongoDB pour lire sur le maître ou sur un esclave, en spécifiant une préférence de lecture. Le tableau 3-3 liste les préférences de lecture supportées par les pilotes MongoDB depuis la version 2.2.

Tableau 3-3. Préférence de lecture en MongoDB

Préférence de lecture	Description
`primary`	Lecture sur le maître. Une erreur est générée si le maître est indisponible.
`primaryPreferred`	Lecture sur le maître. Si le maître est indisponible, par exemple si un basculement est en cours, la lecture se fera sur un secondaire.
`secondary`	Lecture sur un secondaire. Une erreur est générée si aucun secondaire n'est disponible.
`secondaryPreferred`	Lecture sur un secondaire. S'il n'y a pas de secondaire, la lecture se fera sur le maître.
`nearest`	Lecture sur le membre le plus proche, selon les déductions du client. Le client identifie le membre le plus proche à l'aide d'un ping régulier sur les membres.

Voici un exemple de connexion avec PyMongo, le pilote Python de MongoDB.

```
# -*- coding: utf-8 -*-
import sys
from pymongo import MongoReplicaSetClient
from pymongo.errors import ConnectionFailure
from pymongo.read_preferences import ReadPreference

def main():
    try:

        cn = MongoReplicaSetClient (
            "localhost:15000",
            replicaSet='ReplicaSet1',
            read_preference=ReadPreference.SECONDARY,
            tag_sets=[{'dc': 'paris'}, {'dc': 'lille'}]
        )

        print "connexion réussie"
        cn.read_preference = ReadPreference.SECONDARY_PREFERRED

    except ConnectionFailure, e:
        sys.stderr.write(«Erreur de connexion à MongoDB:%s»% e)
        sys.exit(1)

if __name__ == "__main__":
    main()
```

Ici, après avoir importé de PyMongo les modules `MongoReplicaSetClient` et `ReadPreference`, nous ouvrons une connexion à MongoDB à l'aide de la classe `MongoReplicaSetClient`, à qui nous envoyons le nom de l'ensemble de réplicas (ici, nous avons nommé notre ensemble `ReplicaSet1)`

et nous indiquons la valeur de `read_preference`. Nous utilisons aussi la capacité d'indiquer des tags pour choisir quel serveur privilégier dans le jeu de réplicas. Nous étudierons cette fonctionnalité dans le chapitre 8 dédié à MongoDB. Plus loin dans le code, nous modifions `read_preference` pour lui attribuer la valeur `ReadPreference.SECONDARY`. Vous pouvez donc agir ensuite sur `read_preference` pour modifier l'état de la connexion.

Attention à Pymongo

Selon votre version de MongoDB et de Pymongo, le code précédent peut changer, et ce, pour deux raisons. Depuis la version 2.6 de MongoDB a été créé un nouvel objet de connexion qui est à utiliser dans vos nouveaux développements: `mongoClient` au lieu de `connection`. De son côté, Pymongo s'est mis à jour avec deux objets: `MongoClient` et `MongoReplicaSetClient`. Ensuite, à partir de la version 3 de Pymongo, qui doit sortir en 2015, l'objet `MongoReplicaSetClient` va disparaître pour être fusionné avec `MongoClient`. La présence de deux objets distincts résulte d'une mauvaise décision de la part des développeurs du pilote, comme ils l'expliquent dans cette entrée de blog: http://emptysqua.re/blog/good-idea-at-the-time-pymongo-mongoreplicasetclient/.

Le sharding

Le nom anglais *shard* signifie « éclat » (comme un éclat de verre) ou « tesson » (de bouteille ou de poterie). Le terme dérivé *sharding* décrit donc l'éclatement des données en partitions distribuées sur plusieurs machines, de façon aussi automatique que possible par le moteur NoSQL. Ce terme est utilisé dans beaucoup de moteurs, et est parfois disponible par des additions aux moteurs qui ne le supportent pas nativement. Par exemple, au moins trois extensions sont disponibles pour effectuer du sharding sur CouchDB : BigCouch, Lounge et Pillow. Par ailleurs, Couchbase Server, basé sur CouchDB, permet le sharding.

Le sharding consiste donc en un partitionnement des données, ou clustering (à ce niveau, les concepts ont tendance à se chevaucher). Le terme spécifique de sharding est né pour désigner une forme de partitionnement présentant des caractéristiques particulières.

- L'éclatement est géré automatiquement par le système. Il n'y a, a priori et autant que possible, aucun besoin de le gérer manuellement.
- L'éclatement est par nature distribué. Cela n'a pas de sens de créer plusieurs shards sur la même machine, en tout cas pas en production.
- La répartition de charge est possible.

Implémentations relationnelles

Le concept de sharding n'est pas limité aux bases NoSQL. Il s'agit en définitive d'un partitionnement de données effectué sur une clé, ce qui ne pose pas de problème majeur pour un SGBDR. Par exemple, SQL Azure Federations est une implémentation du concept de sharding sous SQL Azure, la base de données relationnelle de Microsoft sur son cloud. Dans SQL Azure Federations, chaque base de données réside sur un serveur et contient une plage de clés. Les bases sont fédérées ensemble et représentent l'intégralité des données. Le modèle est le suivant : *Federation Root* → *Member* → *Atomic Unit (AU)* → *Table*. L'Atomic Unit correspond à un shard.

En soi, le sharding peut aussi être classé dans la section suivante, dédiée à la distribution sans maître. MongoDB est un exemple de moteur qui implémente le sharding basé sur une machine principale qui sait comment sont distribué les shards, et oriente le client. Nous verrons au chapitre 8 comment mettre en place concrètement le sharding sur MongoDB.

La distribution sans maître

La problématique d'un système distribué avec maître est bien entendu la présence d'un SPOF *(Single Point Of Failure)*, c'est-à-dire d'un élément non redondant qui représente un risque de défaillance pour tout le système. Les moteurs NoSQL qui s'inspirent du modèle BigTable de Google sont bâtis autour d'un système avec maître. D'autres moteurs comme Redis ou CouchDB ne sont pas nativement distribués, la question ne se pose donc pas. Des extensions permettent toutefois de les faire évoluer vers la distribution des données. Quant aux moteurs inspirés de Dynamo d'Amazon, ils se basent sur une architecture sans maître, où théoriquement toute machine a la même importance et les mêmes capacités dans le cluster. L'objectif est de pouvoir assurer un service complet en se connectant à n'importe quelle machine, même si plusieurs membres du cluster tombent.

Pour bâtir un système distribué et décentralisé fonctionnel, il faut répondre à plusieurs questions : comment diriger les requêtes des clients sur les machines qui contiennent l'information souhaitée ? Et comment maintenir l'état du système, notamment comment savoir quelles sont les machines qui composent le cluster ? Il faut aussi savoir où et comment distribuer les données, de façon à ce qu'elles soient réparties autant que possible et qu'elle puissent être répliquées sur différents sites. Dans la section suivante, nous allons aborder quelques concepts et technologies qui répondent à ces questions.

La table de hachage distribuée

La distribution des données implémentée par Dynamo et décrite dans le document dont nous avons parlé est basée sur le modèle de la table de hachage distribué (DHT, *Distributed Hash Table*). Le principe de la DHT est utilisé dans les outils d'échange de fichiers comme Gnutella ou BitTorrent. Prenons l'exemple de BitTorrent. À la base, ce protocole n'est pas tout à fait décentralisé. Certes, l'échange de fichiers s'opère selon un mode P2P *(peer-to-peer)*, mais la détection de pairs qui détiennent le fichier passe par un serveur, nommé tracker. Tout téléchargement de fichier par BitTorrent doit, dans le protocole d'origine en tout cas, passer au préalable par la communication avec un tracker, donc un serveur maître. Pour contourner cette problématique, le client BitTorrent officiel implémente une DHT, dont le format est maintenant reconnu par la plupart des clients BitTorrent. Cette DHT maintient sur chaque client BitTorrent une partie de l'information d'attribution des fichiers, réservée normalement au tracker. En interrogeant d'autres clients, un client BitTorrent peut retrouver des pairs.

Une DHT est représentée conceptuellement par un espace de clés (keyspace), qui correspond à une grande table de hachage. Cet espace de clés est distribué sur les nœuds du système. Dynamo, Cassandra, Riak, etc. sont des formes de tables de hachage distribuées, qui maintiennent leurs données dans une table scalable de paires clé-valeur réparties sur un nombre potentiellement

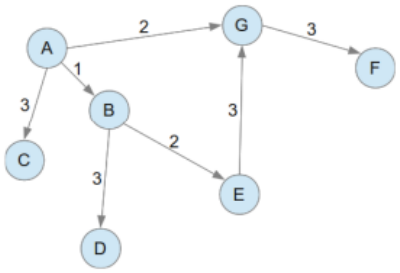
infini de nœuds. La connaissance de la topologie du cluster est également distribuée sur chaque noeud du cluster : on peut donc parler d'un système de stockage en P2P.

Le protocole de bavardage

Un protocole de bavardage *(gossip protocol)* est une méthode de distribution des informations virales basée sur ce qu'on pourrait appeler la théorie des épidémies (théorisée par Norman Bailey dans son livre *The Mathematical Theory of Infectious Diseases*, 2nd Edition, Hafner Press/MacMillian Pub. Co., 1975). En résumé, si une personne est contagieuse, elle va entrer en contact avec des sujets réceptifs qui vont tomber malades et eux-mêmes contaminer d'autres sujets, ce qui a un effet de croissance exponentielle. Les systèmes distribués peuvent utiliser le protocole de bavardage pour disséminer des informations, comme la disponibilité ou l'ajout d'un nœud dans le système ou la modification d'une donnée.

Dans un protocole de bavardage, les nœuds engagent périodiquement une conversation avec un nombre choisi de machines (des pairs) pour échanger des informations (d'un à trois nœuds toutes les secondes en Cassandra, par exemple). Après un certain temps, une information connue d'un seul nœud au départ sera communiquée à tous les nœuds. Comme l'échange d'informations ne se fait pas en temps réel, mais selon une forme de planification « relâchée », le réseau n'est pas submergé par les communications entre les nœuds, ce qui le maintient en bonne santé. Cela implique que la distribution totale de l'information va prendre du temps. Il ne s'agit donc pas d'une solution valable si on souhaite que l'information soit disponible pour tous le plus rapidement possible. La figure 3-4 montre de façon schématique le fonctionnement d'un protocole de bavardage.

Figure 3-4
Protocole de bavardage



Ici, le nœud A a une information à communiquer au système. Il va entrer en contact avec un nœud choisi au hasard parmi les nœuds dont il a connaissance. C'est la communication numéro 1 avec le nœud B. Cette communication s'établit dans les deux sens : si le nœud B a aussi une information à communiquer, il l'enverra à A durant leur conversation. Ensuite, à intervalles réguliers, chaque nœud en contactera un autre. A contacte G et B contact E. Ensuite, A contacte C, B contacte D et E contacte G. Il se trouve que G a déjà reçu l'information de la part de A, ce sont les risques du bavardage : si je colporte une rumeur, je vais aussi la raconter à des gens qui la connaissent déjà.

Dans le document décrivant Dynamo (http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html), Amazon indique que le protocole de bavardage lui sert à propager les changements de nœuds

du cluster et à maintenir une vision à consistance finale de cette appartenance. Si un nouveau nœud est ajouté, il commence à publier sa présence par bavardage et après quelque temps, tous les nœuds en seront avertis par échange d'un historique des changements de membres du cluster. Le bavardage sert aussi, dans Dynamo, à l'échange des informations de partitions, c'est-à-dire la distribution des clés entre les machines (*consistent hash space*, voir section suivante), ce qui fait que chaque nœud sait quelle est la rangée de clés gérée par tous les autres nœuds du cluster. Cela permet donc à chaque machine de recevoir des demandes de lecture ou d'écriture de la part d'un client, et de transmettre cette demande à un nœud qui gère les bonnes données. Riak, un moteur NoSQL décentralisé qui s'inspire fortement de Dynamo, utilise le protocole de bavardage pour maintenir l'état du cluster. Toute machine qui joint ou quitte le cluster va lancer un message par bavardage. Les nœuds échangent également régulièrement leurs informations d'appartenance, de façon à s'assurer qu'aucun nœud n'ait manqué une mise à jour.

Le protocole de bavardage est aussi utilisé pour gérer la cohérence finale, comme nous le verrons plus loin dans la section consacrée à la cohérence finale par néguentropie, page 88.

Le hachage consistant

Bâtir un système distribué implique également de partitionner les données pour les distribuer de la façon la plus équitable possible entre les nœuds. A priori, le moyen le plus logique pour réaliser cela est de se baser sur la clé (primaire) de la donnée (par exemple, l'identifiant du document), et de trouver un moyen, à partir de cette clé, de distribuer les données. Dans un système sans maître, sans coordinateur, il faut aussi que les clients puissent être redirigés sur les machines qui contiennent les données voulues. Une réponse basique à ces problématiques est la distribution des clés avec un modulo. En général, il s'agit d'un hachage de la clé à l'aide d'un algorithme qui assure une bonne distribution du résultat et retourne une valeur entière, par exemple la méthode `hashCode()` qu'on trouve dans toute classe Java et qui retourne un entier 32 bits. Pour notre démonstration, nous allons travailler directement avec un entier. Cette solution est illustrée par la figure 3-5.

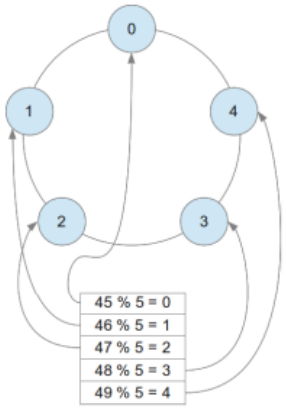


Figure 3-5
Distribution par hashcode

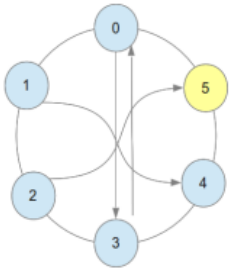
Ici, nous avons bâti un système distribué sur cinq nœuds. Nous voulons distribuer nos données de la façon la plus régulière possible. Notre algorithme d'attribution des données est basé sur un modulo : lorsque nous créons un nouvel enregistrement, nous calculons un modulo 5 par rapport à la valeur de l'identifiant. Le reste de la division par cinq nous retourne le numéro du nœud auquel nous allons attribuer l'enregistrement. L'attribution au bon serveur peut être résumée par la ligne de code suivante :

```
server = serverlist[ hash(key)% len(serverlist) ]
```

Et par exemple, l'enregistrement portant l'identifiant 48 sera envoyé sur le nœud 3. C'est aussi simple que cela. Et, à la rigueur, si le client est au courant du nombre de nœuds, il peut deviner lui-même où se trouve l'enregistrement qu'il cherche, car il connaît la clé, calcule le modulo et va directement sur le bon serveur. Mais en pratique, cette connaissance du client pose problème, car cela implique de stocker trop de données de configuration du côté du client, et cela fige le système : comment, par exemple, faire évoluer la configuration du client quand on ajoute des nœuds ?

Justement, que va-t-il se passer lorsqu'on va souhaiter ajouter un nœud ? Nous devons modifier notre modulo pour qu'il corresponde au nouveau nombre de nœuds, mais cela implique que l'attribution d'une majorité des enregistrements doit changer, comme illustré sur la figure 3-6.

Figure 3-6
Ajout d'un nœud dans une distribution par hashcode



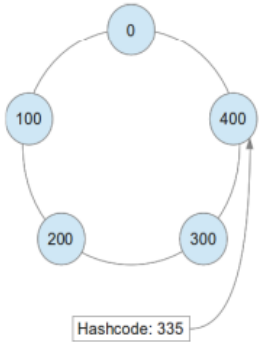
0 → 45 % 6 = 3
1 → 46 % 6 = 4
2 → 47 % 6 = 5
3 → 48 % 6 = 0
4 → 49 % 6 = 1

Cela veut donc dire qu'à l'ajout d'un nœud, une réorganisation des données doit se faire sur tous les nœuds. C'est un défaut majeur, et sur un système de grande envergure, c'est même pratiquement inenvisageable. C'est pour éviter cette problématique que la technique du *consistent hashing* a été développée.

L'idée du hachage consistant est d'attribuer des rangées de valeurs pour chaque nœud. Conceptuellement, le système distribué est vu comme un anneau (*hash ring*, anneau de hachage), où chaque nœud occupe une position sur l'anneau. Chaque nœud se voit attribuer une valeur de hachage, qui définit la rangée de valeurs qui est hébergée. L'algorithme de hachage calcule un *hash code* à

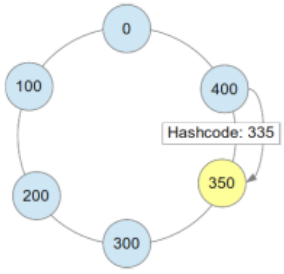
partir de la clé, trouve le nœud dont la valeur est immédiatement supérieure et y stocke la donnée. Un nœud contient donc toutes les clés inférieures à sa valeur de rangée et supérieures à la valeur du nœud précédent, comme illustré sur la figure 3-7.

Figure 3-7
Distribution par hachage consistant



Lorsqu'un nœud est ajouté à l'anneau, il prend une valeur de rangée, et donc scinde une rangée existante. L'attribution des nœuds ne sera donc troublée que pour une seule rangée, et un nombre peu important de données sera déplacé, comme illustré sur la figure 3-8.

Figure 3-8
Ajout d'un nœud avec hachage consistant



Choix du nœud à scinder

Quand le système connaît le nombre de clés qui sont gérées par chaque nœud, l'attribution de la rangée d'un nouveau nœud peut se faire automatiquement en scindant le nœud le plus chargé. C'est par exemple le cas avec Apache Cassandra si vous utilisez le hachage consistant dans votre cluster.

L'une des premières utilisations du hachage consistant dans un cadre NoSQL est documentée par le site de partage musical last.fm dans une entrée de blog datant de 2007 (http://www.lastfm.fr/user/RJ/journal/2007/04/10/rz_libketama_-_a_consistent_hashing_algo_for_memcache_clients). La problématique de distribution des données que nous venons d'illustrer se posait pour eux sur

les serveurs memcached. Afin d'éviter de redistribuer toutes les clés sur leurs serveurs de cache, last.fm a développé une bibliothèque en C implémentant un algorithme de hachage consistant qu'ils ont appelé Ketama. La bibliothèque s'appelle libketama, son code source est disponible sur un serveur Subversion à l'adresse suivante : http://www.audioscrobbler.net/development/ketama/. Le hash ring y est nommé le continuum.

Une implémentation simple de hachage consistant pour Python est disponible dans le module `hash_ring` (http://pypi.python.org/pypi/hash_ring/), qui peut être utilisé pour gérer l'attribution de paires clé-valeur à des serveurs memcached. Il est documenté sur cette page de blog : http://amix.dk/blog/post/19367.

Exemple de hachage consistant : Redis

Conformément à l'esprit du logiciel libre, les moteurs NoSQL n'ont pas encore implémenté toutes les fonctionnalités voulues. Il est cependant possible de les ajouter soi-même, ou de chercher un peu pour trouver une bonne âme ayant déjà fait le travail pour nous. L'engouement du monde NoSQL fait qu'il génère beaucoup de développements et d'ajouts. Nous allons en donner un exemple avec un module Python, nommé `redis-shard`. Ce dernier implémente une API de sharding basée sur du hachage consistant pour distribuer une base de données Redis sur plusieurs machines. Pour l'instant, la clusterisation native de Redis est en développement (les spécifications sont disponibles à cette adresse : http://redis.io/topics/cluster-spec). En attendant, une approche implémentée au niveau du client peut très bien faire l'affaire. Le concept peut s'appeler du *presharding* (http://oldblog.antirez.com/post/redis-presharding.html). Il s'agit simplement d'ajouter une couche au niveau de l'application client qui distribue les données sur une liste de serveurs Redis, afin de décider avant l'envoi de la bonne destination. Le code source de `redis-shard` est disponible sur le site github.com : https://github.com/youngking/redis-shard. La configuration se fait en définissant une liste de serveurs disponibles. Par exemple :

```
from redis_shard.shard import RedisShardAPI
servers = [
    {'name':'server1','host':'192.1.68.30.10','port':6379,'db':0},
    {'name':'server2','host':'192.1.68.30.20','port':6379,'db':0},
    {'name':'server3','host':'192.1.68.30.30','port':6379,'db':0},
    {'name':'server4','host':'192.1.68.30.40','port':6379,'db':0}
]
```

Nous allons simplement reproduire ici une partie simplifiée du code source, situé dans le fichier `hashring.py` (https://github.com/youngking/redis-shard/blob/master/redis_shard/hashring.py), qui montre comment réaliser un hachage consistant.

```
import zlib
import bisect

class HashRing(object):
    """Consistent hash for nosql API"""

    def __init__(self, nodes=[], replicas=128):
        """Manages a hash ring.
```

```
        self.nodes = []
        self.replicas = replicas
        self.ring = {}
        self.sorted_keys = []

        for n in nodes:
            self.add_node(n)

    def add_node(self, node):
        self.nodes.append(node)
        for x in xrange(self.replicas):
            crckey = zlib.crc32("%s:%d"% (node, x))
            self.ring[crckey] = node
            self.sorted_keys.append(crckey)

        self.sorted_keys.sort()

    def get_node(self, key):
        n, i = self.get_node_pos(key)
        return n

    def get_node_pos(self, key):
        if len(self.ring) == 0:
            return [None, None]
        crc = zlib.crc32(key)
        idx = bisect.bisect(self.sorted_keys, crc)
        idx = min(idx, (self.replicas * len(self.nodes)) - 1)
                  # prevents out of range index
        return [self.ring[self.sorted_keys[idx]], idx]

    def iter_nodes(self, key):
        """Given a string key it returns the nodes as a generator that can hold the
key.
        """
        if len(self.ring) == 0:
            yield None, None
        node, pos = self.get_node_pos(key)
        for k in self.sorted_keys[pos:]:
            yield k, self.ring[k]
```

Le constructeur de la classe `HashRing` remplit trois structures, dont le dictionnaire `ring` et la liste `sorted_keys`. Cette dernière contient les valeurs de hachage qui forment les étapes de l'anneau, et `ring` attribue un nœud à ces étapes. Pour chaque nœud de l'anneau, le code suivant est appelé :

```
for x in xrange(self.replicas):
    crckey = zlib.crc32("%s:%d"% (node, x))
    self.ring[crckey] = node
    self.sorted_keys.append(crckey)
```

À l'instanciation de la classe `HashRing`, le membre `replicas` est initialisé avec une valeur entière. Il permet d'indiquer un nombre de points virtuels afin d'améliorer la distribution des données sur les nœuds. La valeur par défaut de `replicas` est 128. La fonction Python `xrange()`, assez semblable à la fonction `range()`, permet de créer une rangée utilisable dans une boucle. Dans ce cas, `xrange(128)` retourne un objet énumérable de 128 éléments numérotés de 0 à 127. Pour chacun de ces éléments, un `crc` est calculé et attribué à ce nœud. Chaque nœud aura donc 128 hachages qui pointeront sur lui. Le crc *(cyclic redundancy check)* est généré à l'aide de la méthode `crc32` de l'objet `zlib` (fournissant un algorithme de compression compatible `gzip`), qui effectue un calcul de somme de contrôle assez basique et retourne une valeur entière 32 bits. Faisons un petit test de l'utilisation de cette méthode dans l'interpréteur interactif de Python pour nous familiariser avec lui :

```
python
Python 2.7.2+ (default, Jul 20 2012, 22:12:53)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import zlib
>>> import uuid
>>> id = uuid.uuid4()
>>> print id
96adc542-547a-488a-80ec-29c37a6d6fb7
>>> print zlib.crc32(id.bytes)
321027949
```

Nous avons ici importé les modules `uuid` et `zlib`. Le module `uuid` permet de générer des UUID *(Universally Unique IDentifier)* selon la définition du RFC 4122 (http://www.ietf.org/rfc/rfc4122.txt). Nous créons un UUID (`96adc542-547a-488a-80ec-29c37a6d6fb7`) et nous passons sa représentation en chaîne de 16 octets (à l'aide de la méthode `bytes()` de l'objet `uuid`) à `zlib.crc32()`, ce qui retourne l'entier `321027949`. Simplissime.

Revenons au code de `hashring.py`. La recherche du nœud concernant une clé est assurée par la méthode `get_node()`. Elle appelle la méthode `get_node_pos()` qui calcule le `crc` de la clé et le passe dans la méthode `bisect()` du module `bisect`, qui offre un algorithme de maintenance de tableau trié. `bisect` permet d'ajouter des éléments à un tableau trié sans avoir à le trier à nouveau dans son intégralité à chaque fois. La méthode `bisect` utilisée dans la ligne :

```
idx = bisect.bisect(self.sorted_keys, crc))
```

emploie un algorithme de division *(bisection)* pour retourner un point d'insertion directement à droite (après) de la valeur de `crc` dans la liste `self.sorted_keys`, qui est la liste des valeurs de `crc` attribuées à chaque nœud dans la configuration du cluster `redis-shard`. C'est donc exactement ce qu'il nous faut. L'index est ensuite construit dans la ligne :

```
idx = min(idx, (self.replicas * len(self.nodes)) - 1)
```

La fonction `min()` retourne la plus petite valeur de la liste, c'est-à-dire soit l'index retrouvé avec `bisect`, soit la dernière valeur d'index si celui-ci dépasse le nombre de points (nombre de points virtuels × nombre de nœuds), le but étant évidemment de gérer les valeurs de `crc` qui sont au-delà

de la dernière valeur du tableau de `sorted_keys`. L'adresse du nœud est ensuite retournée en le cherchant dans le dictionnaire `ring` :

```
return [self.ring[self.sorted_keys[idx]], idx]
```

Nous pensions qu'il était intéressant de vous montrer un algorithme simple de hachage consistant pour vous permettre de mieux appréhender la méthodologie. Comme vous le voyez, à l'aide d'un langage riche comme Python, l'implémentation de hachage consistant est aisée et claire.

Les VNodes

Le hachage consistant est une bonne réponse aux problématiques d'élasticité d'une distribution des données de type DHT. Toutefois, il ne s'agit pas forcément de la solution la plus performante dans tous les cas. Une autre méthode existe, qui consiste à découper les données en nœuds virtuels, et à gérer la réorganisation des données lors d'un redimensionnement du cluster par nœud virtuel. Le terme VNode traditionnel est utilisé dans les systèmes de fichiers UNIX pour désigner un mécanisme d'abstraction du système de fichiers, mais il est maintenant utilisé dans un sens différent par Cassandra, et nous avons décidé dans cet ouvrage de reprendre le terme spécifique de Cassandra pour parler de la méthodologie sous-jacente.

Le hachage consistant présente deux problèmes. Tout d'abord, l'ajout d'un nœud va couper les données d'un seul autre nœud, ce qui ne contribue pas à l'équilibre général du cluster. Tout au plus, cela peut rééquilibrer un seul nœud, ou réduire la charge relative de ce nœud et du nouveau par rapport aux autres. L'idéal serait plutôt une méthode qui maintienne un équilibre général.

Par ailleurs, l'échange de données entre les deux machines va fortement charger le nœud coupé, qui sera donc sollicité en exclusivité, et le trafic réseau entre ces deux machines subira un pic. Il faudrait un moyen qui, sans provoquer une migration aussi importante que dans la technique du modulo, permette aussi de distribuer la charge de l'alimentation du nouveau nœud. Cette technique, c'est celle des nœuds virtuels. Elle est maintenant utilisée par Cassandra en remplacement du hachage consistant, et elle est également présente dans Couchbase Server.

La base de données va être découpée, non pas en segments qui correspondent à des machines réeelles, mais en beaucoup plus de blocs, qui seront répartis sur les nœuds. Chaque bloc représentera donc une unité et chaque nœud physique va héberger un nombre proche de blocs. Lors de l'ajout d'un nouveau nœud, chaque machine va choisir un certain nombre de ses blocs pour les envoyer sur le nouveau membre. Cela permet d'assurer un bien meilleur équilibre et de répartir la charge réseau lors de la réorganisation du cluster. Cela ne représentera pas de charge inutile, car seul le bon nombre de blocs sera déplacé.

Réplication avec prise en compte de la topologie du réseau

Dans la section consacrée à la distribution maître-esclave page 72, nous avons abordé le sujet de la réplication maître-esclave et du sharding. Dans un système sans maître, comme les moteurs basés sur Dynamo, la distribution a les mêmes contraintes, qui sont finalement la transposition à l'échelle d'un système distribué des concepts d'un système RAID : miroir des données pour assurer une redondance, et *striping* pour optimiser les lectures-écritures. Si nous prenons l'exemple de Cassandra, les données y sont non seulement distribuées à l'aide d'un algorithme

de hachage consistant, mais elles sont aussi répliquées sur d'autres nœuds d'une façon très configurable. Comme nous venons de le voir, la DHT de Cassandra maintient l'information du nœud qui contient les données, mais aussi des autres nœuds sur lesquels ces données sont répliquées, et ces informations sont échangées par bavardage. Mais Cassandra permet aussi d'indiquer une stratégie de duplication des données incluant des paramètres de dispersion géographique, nommée le *snitch* (terme d'argot signifiant la balance ou l'informateur). Ce snitch est à indiquer dans le fichier `cassandra.yaml` (situé dans le répertoire `/etc/cassandra/`) sur le système, dans l'option `endpoint_snitch`. On attribue comme valeur de snitch une classe qui implémente l'interface `IendpointSnitch` dans Cassandra. On peut donc créer sa propre règle de réplication des données. Les classes existantes sont listées dans le tableau 3-4.

Tableau 3-4. Snitches dans Cassandra

Classe	Description
`SimpleSnitch`	Dans un déploiement sur un seul data center, cette classe privilégie la proximité pour copier des réplicas.
`PropertyFileSnitch`	La proximité est définie par valeur de rack et de data center. Ces informations doivent être saisies dans le fichier `cassandra-topology.properties`.
`GossipingPropertyFileSnitch`	L'identification du rack et du data center local est réalisée par inscription dans le fichier `cassandra-rackdc.properties`. Cette information est ensuite communiquée au cluster par bavardage.
`RackInferringSnitch`	La notion de rack et de data center est déduite de l'adresse IP du nœud. Le rack correspond au troisième octet de l'IP et le data center au deuxième octet.
`Ec2Snitch`	Cette classe peut être utilisée pour le déploiement sur un cloud Amazon EC2 dans une seule région.
`Ec2MultiRegionSnitch`	Cette classe peut être utilisée pour le déploiement sur un cloud Amazon EC2 dans plusieurs régions.

Pour l'option `PropertyFileSnitch`, un fichier de configuration de la topologie `cassandra-topology.properties` à placer dans le répertoire de configuration, pourrait ressembler à ceci :

```
# Data Center 1

192.168.10.10=DC1:RAC1
192.168.10.11=DC1:RAC1
192.168.10.12=DC1:RAC1

192.168.10.20=DC1:RAC2
192.168.10.21=DC1:RAC2
192.168.10.22=DC1:RAC2

# Data Center 2

192.168.20.10=DC2:RAC1
192.168.20.11=DC2:RAC1
192.168.20.12=DC2:RAC1
```

```
192.168.20.20=DC2:RAC2
192.168.20.21=DC2:RAC2
192.168.20.22=DC2:RAC2

# une valeur par défaut pour les nœuds inconnus
default=DC3:RAC1
```

Le fichier attribue les adresses IP des serveurs à des positions de rack et de data center (DC). Les noms de rack et de data center doivent ensuite être définis dans la stratégie de placement des espaces de clés de Cassandra, lorsqu'on en définit une.

Définir la stratégie de placement

La stratégie de placement était auparavant définie dans le fichier de configuration `cassandra.yaml`. Depuis la version 0.7.4, elle est précisée dans le système d'espaces de clés *(keyspace system)* de Cassandra et à la création d'un espace de clés.

Par défaut, deux stratégies de placement sont possibles dans Cassandra, comme indiqué dans le tableau 3-5.

Tableau 3-5. Stratégies de placement

Stratégie de placement	Description
org.apache.cassandra.locator.SimpleStrategy	Simple, gérée automatiquement par Cassandra.
org.apache.cassandra.locator.NetworkTopologyStrategy	Prend en compte la topologie définie dans `cassandra-topology.properties`.

La définition de la stratégie `SimpleStrategy` permet d'indiquer un nombre de réplicas à maintenir pour l'espace de clés. Par exemple :

```
CREATE KEYSPACE passerelles
WITH strategy_class = 'SimpleStrategy'
AND strategy_options:replication_factor = 3;
```

crée l'espace de clés `passerelles` en spécifiant que Cassandra doit maintenir trois réplicas des données. La stratégie `NetworkTopologyStrategy` permet de spécifier les data centers spécifiés dans `cassandra-topology.properties` :

```
CREATE KEYSPACE passerelles
WITH strategy_class = 'NetworkTopologyStrategy'
AND strategy_options:DC1 = 2 AND strategy_options:DC2 = 2;
```

Ce code crée l'espace de clés `passerelles` en spécifiant que Cassandra doit maintenir deux réplicas sur DC1 et deux réplicas sur DC2.

Cassandra est l'un des moteurs NoSQL dont les possibilités de placement des réplicas sont les plus affinées. Riak, dans sa version Entreprise, est un autre exemple de moteur NoSQL qui offre une fonctionnalité de réplication multi-data center.

La cohérence finale

Dans le chapitre précédent, nous avons abordé les problématiques de cohérence transactionnelle et d'ACIDité de la transaction. Nous avons vu également que le théorème CAP affirme, en un mot, qu'il est impossible de maintenir une transaction cohérente et une bonne disponibilité dans un système distribué. Il faut sacrifier une partie de l'un des trois critères pour augmenter les deux autres. Le choix le plus souvent effectué est d'abandonner la cohérence transactionnelle forte.

La réponse à la cohérence dans un système distribué, telle qu'elle se présente dans les moteurs NoSQL à distribution asynchrone, s'appelle la cohérence finale *(eventual consistency)*. Nous avons vu qu'un système ACID ne rend pas l'état incohérent disponible et qu'il bloque ou protège l'accès aux ressources sales. La cohérence finale laisse le système ouvert, permet la lecture et s'occupe au fur et à mesure de diffuser les modifications sur tous les nœuds, en mettant en place des outils de réconciliation en cas d'incohérence. Il s'agit d'une forme de verrouillage optimiste.

BASE

À travers l'adoption du théorème CAP comme un axiome indépassable de la gestion transactionnelle d'un serveur de données, un jeu de mot a été créé pour définir l'autre choix, à savoir la cohérence finale. Il s'agit du concept BASE, pour *Basically Available, Soft state, Eventually consistent.*

Quand une requête arrive dans un moteur NoSQL, elle n'est pas forcément sauvegardée immédiatement. Elle peut être conservée en mémoire ou être appliquée sur une machine sans être immédiatement distribuée sur d'autres réplicas. La propagation des mises à jour s'effectuera au fur et à mesure.

Exemple de la propagation des DNS

Le mécanisme de propagation des DNS (*Domain Name Server*, serveur de nom de domaine Internet) est souvent cité pour illustrer un système de cohérence finale qui fonctionne. Lorsque vous voulez changer l'adresse de la machine sur laquelle pointe une URL (vous indiquez, par exemple, que http://www.babaluga.com pointe maintenant sur l'adresse IP 88.191.70.21), vous modifiez cette adresse dans un ou plusieurs DNS et vous indiquez auprès du registrar du domaine que ces DNS sont autoritaires pour ce domaine.

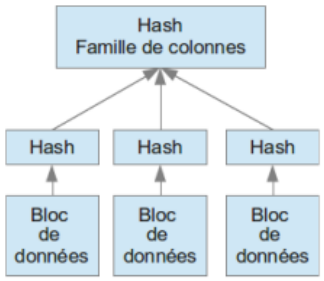
Lorsqu'une machine sur Internet doit trouver l'adresse IP correspondant à http://www.babaluga.com, elle demande la correspondance à son DNS, qui va répercuter la recherche à une hiérarchie de serveurs DNS pour trouver le serveur autoritaire et retrouver l'adresse. Comme ceci peut prendre du temps, les serveurs DNS maintiennent un cache local qui leur permet de répondre plus rapidement. C'est ce cache qui est potentiellement déconnecté de la réalité la plus récente. Ce cache se rafraîchit régulièrement, c'est-à-dire que ses entrées sont invalidées après un temps nommé, le *Time To Live* (TTL). À un instant donné, une requête sur un DNS peut donc retourner des résultats obsolètes, mais à terme (après cinq jours, au maximum), tous les DNS mondiaux verront leur cache régénéré. Notons enfin que même si on appelle traditionnellement ce fonctionnement une propagation, il s'agit ici plutôt d'un rafraîchissement de cache en mode pull que d'une propagation en mode push.

La néguentropie et l'arbre de Merkle

Dans les moteurs NoSQL à distribution décentralisée et asynchrone dont l'architecture s'inspire de Dynamo, comme Cassandra ou Riak, le protocole de bavardage est aussi utilisé pour réconcilier les données de façon à ce que tous les réplicas obtiennent à terme la version la plus actuelle des données. C'est un système néguentropique (dit aussi anti-entropique, de l'anglais *anti-entropy*), qui permet la cohérence finale. On parle de système néguentropique en référence au terme inventé par le physicien Erwin Schrödinger pour décrire la capacité des êtres vivants à maintenir un ordre malgré la tendance naturelle de l'univers à l'entropie. Cette réconciliation et organisation des données s'appuie sur la technique de l'arbre de Merkle (*Merkle Tree*, aussi appelé *Hash Tree*), qui est un mécanisme de simplification de comparaison d'un volume de données. Le concept a été créé par Ralph Merkle, chercheur en cryptographie et en nanotechnologie, et breveté en 1982.

Imaginez que vous souhaitiez comparer deux bases de données qui sont supposées être de parfaits réplicas. Vous allez probablement le faire à l'aide d'un algorithme de hachage, un *checksum*, par exemple en MD5, qui va produire un résumé d'une valeur plus longue. Vous allez produire un checksum de chaque ligne de chaque table, et comparer ces checksums avec les mêmes provenant de l'autre base de données. Ainsi présentée, l'opération peut sembler longue et coûteuse. Pour diminuer le travail, vous allez probablement calculer le hachage d'une table toute entière et effectuer la comparaison sur cette seule valeur. Si les deux hachages correspondent, cela signifie que votre table est identique. En revanche, si vous constatez une différence, vous pourrez descendre dans les détails en créant des hachages par pages de données, pour identifier les pages qui ont changé, et ensuite en créant des hachages de chaque ligne dans les pages identifiées, pour retrouver les lignes modifiées. C'est le principe simple de l'arbre de Merkle, beaucoup utilisé dans les systèmes d'échange de données en pair-à-pair, pour valider la cohérence d'un téléchargement. Ce principe est schématisé, pour un système comme Cassandra, sur la figure 3-19.

Figure 3-9
L'arbre de Merkle

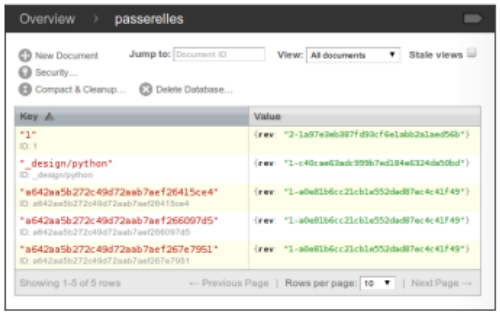


Il reste bien sûr à trouver le moyen de calculer cet arbre de Merkle au bon moment, en minimisant l'impact sur les performances. En Cassandra, par exemple, l'arbre de Merkle est calculé par un validateur durant des opérations de compactage (voir la documentation disponible à l'adresse suivante : http://wiki.apache.org/cassandra/ArchitectureAntiEntropy).

Multi-Version Concurrency Control

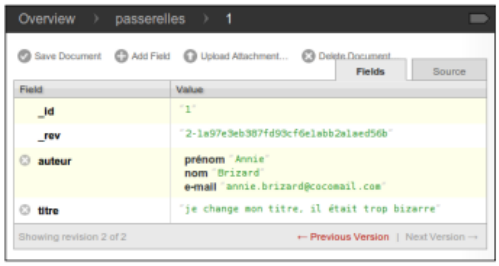
Il reste un problème : comment gérer les conflits ? Dans un système distribué, on peut mettre à jour la même donnée répliquée sur deux machines différentes en même temps. Comment prendre en compte ce cas de figure ? La réponse passe souvent par la gestion de versions. Si nous prenons l'exemple de CouchDB, il inclut un mécanisme de gestion de versions (*Multi-Version Concurrency Control*, MVCC) permettant de gérer un verrouillage optimiste. Nous en verrons les détails dans la partie II de cet ouvrage, « Panorama des principales bases de données NoSQL », mais esquissons-en déjà ici les principes de base. Chaque fois que vous enregistrez un document dans CouchDB, un numéro de version est automatiquement attribué par CouchDB, et visible dans un élément JSON nommé `_rev`, présent dans tous les documents. Il s'agit d'un UUID, comme illustré sur la figure 3-10 représentant l'interface web de CouchDB, nommée Futon.

Figure 3-10
Futon



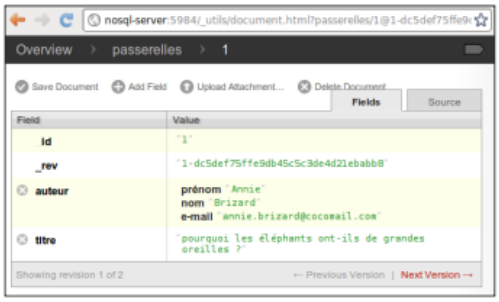
Vous voyez ici cinq documents, la clé est indiquée dans la liste de gauche, et la liste de droite affiche le dernier numéro de révision. Il s'agit bien d'un UUID, auquel CouchDB ajoute un compteur de version en préfixe. Ainsi, le document 1 a été modifié, puisque son numéro de révision commence par 2-. Si nous entrons dans le document, nous voyons qu'il est possible de naviguer dans les versions, les liens de navigation sont visibles en bas de la fenêtre (figures 3-11 et 3-12).

Figure 3-11
Parcourir les versions dans Futon



Si nous cliquons sur le lien Previous Version, nous affichons la version précédente, bien sûr. Notez le 1- au début du numéro de révision et l'adresse de la page Futon visible dans la barre d'adresse du navigateur, qui indique en paramètre le numéro de version

Figure 3-12
Version précédente dans Futon



Ainsi, dans CouchDB, la cohérence est maintenue par les versions. Comme CouchDB est nativement non distribué, cette version sert surtout à assurer un verrouillage optimiste. Lorsque vous mettez à jour un document, vous devez indiquer le numéro de version que vous actualisez. Si le serveur CouchDB reçoit une demande de mise à jour d'un document qui n'est pas la dernière version de ce qu'il a dans sa base, il refusera la mise à jour avec une erreur. C'est donc une forme de gestion de concurrence

Utilisation interne des versions

Il faut ici préciser un point : on pourrait se dire qu'il serait intéressant de profiter de cette gestion automatique des versions pour bâtir un audit historique des changements sur les documents. C'est bien entendu faisable, mais à la base, ce mécanisme de gestion de versions n'a pas été conçu à destination du client, mais bien pour la gestion interne de la cohérence des données. Les anciennes versions sont de toute façon supprimées lors d'un compactage de la base CouchDB. Si ce n'était pas le cas, les besoins de stockage, qui ne sont déjà pas négligeables pour CouchDB, seraient ridiculement élevés : il faudrait maintenir en permanence un historique de tous les changements sur les documents.

Mais les moteurs distribués peuvent aussi utiliser les versions pour gérer la concurrence finale.

Cohérence par timestamps

Dans le document de Google décrivant sa base de données BigTable (http://research.google.com/archive/bigtable.html), les données y sont décrites comme un dictionnaire indexé par une clé de ligne, une clé de colonne et un horodatage (« The map is indexed by a row key, column key, and a timestamp »). Le modèle est schématisé ainsi :

```
(row:string, column:string, time:int64) string
```

Cela veut dire qu'un horodatage est placé sur chaque couple ligne-colonne, donc chaque cellule. Ainsi, l'horodatage est encore plus complet que celui proposé par CouchDB. On peut considérer

chaque valeur stockée dans BigTable comme référencée dans trois dimensions : la ligne, la colonne et le temps. Le temps est exprimé par une valeur entière, qui représente la notion de *timestamp* d'Unix : un nombre de secondes depuis une date de référence (une *epoch*), qui correspond au 1er janvier 1970 à minuit. Ce timestamp est utilisé par BigTable et ses clones (HBase, Cassandra, Hypertable) pour servir en interne à la résolution de conflits et à la cohérence finale. Il implique une contrainte : tous les nœuds et les clients doivent avoir leur horloge synchronisée, car le client doit indiquer le timestamp lors d'une mise à jour. Si plusieurs clients mettent à jour la même donnée, c'est le timestamp le plus récent qui verra son écriture validée sur tous les nœuds. Dans Cassandra par exemple, où le timestamp est toujours indiqué par le client qui transmet l'instruction de mise à jour des données, cela peut se révéler un réel problème.

Timestamp et horloge

On sait qu'une horloge d'ordinateur ne peut avoir la même fréquence que l'univers (on ne va pas entrer dans le débat de savoir ce que cela veut dire, surtout si on prend en compte la relativité restreinte). Elle présente en tout cas une forme de dérive *(clock drifting)*. Les seules horloges qui approchent de la dérive zéro sont les horloges atomiques. Chaque horloge d'ordinateur ayant une dérive différente, même si vous définissez à un instant t la valeur d'horloge de toutes vos machines, il se peut que quelques jours plus tard, celles-ci affichent déjà des différences de plusieurs secondes. Il est important dans un moteur NoSQL qui s'inspire de Google BigTable, comme Cassandra, d'assurer une bonne synchronisation de l'horloge.

NTP *(Network Time Protocol)* est le protocole historique à maîtriser dans ce type de système. Il s'agit d'un outil qui va synchroniser l'horloge avec un réseau de serveurs NTP hiérarchique. Des pairs peuvent se synchroniser en référence à des serveurs NTP à un niveau (appelé stratum) supérieur. Les machines de référence auxquelles on se réfère finalement sont typiquement des horloges atomiques ou des GPS (au stratum 0). Les ordinateurs situés au stratum 1 sont reliées directement aux horloges du stratum 0, et ainsi de suite.

NTP doit être installé sur vos systèmes en tant que service. Pour l'installer sur une machine Debian ou Ubuntu, vous pouvez naturellement récupérer le paquet qui se trouve dans les dépôts officiels.

```
sudo apt-get install ntp
```

Vous pouvez ensuite vérifier que le service a bien été démarré.

```
Service ntp status
```

Sa configuration se gère dans le fichier `/etc/ntp.conf`, dans lequel vous indiquez les serveurs de référence, soit des pairs, soit des machines à un stratum supérieur.

NTP est un protocole complexe, notamment parce qu'il intègre des algorithmes sophistiqués. Il nécessite donc un peu d'apprentissage. Pour rester sur les choses qui nous intéressent, disons qu'il maintient un fichier de dérive qu'il est utile de connaître. Sur notre système Ubuntu Server, il se situe à l'adresse `/var/lib/ntp/ntp.drift`. Ce fichier contient l'indication de la dérive de l'horloge de la machine telle qu'observée par NTP. C'est une valeur exprimée en PPM, ou *Part Per Million*, donc un millionième de seconde, donc une microseconde par seconde. Si on fait le calcul, un PPM équivaut à une dérive de 86,4 millisecondes par jour.

Sur un système UNIX, la commande suivante vous permet de faire directement le calcul, en utilisant l'outil `bc`, qui est un calculateur en ligne de commande.

```
echo 'cat /var/lib/ntp/ntp.drift'*86.4 | bc
```

Sur notre machine, la valeur est –1398.556, soit 1,4 seconde par jour, ce qui est significatif pour Cassandra. Imaginez qu'au bout d'une semaine vous ayez une différence de dix secondes par exemple entre deux machines : cela suffit à écraser des modifications ultérieures par des modifications précédentes à cause d'une erreur de timestamp.

Pour corriger cette dérive, NTP va régulièrement réaffecter l'horloge système, en paliers dans le cas où la différence avec l'heure de référence est trop grande, pour éviter de provoquer des incohérences dans les applications qui maintiennent un journal, par exemple. En fait, le but de NTP n'est pas forcément de synchroniser, mais plutôt de discipliner l'horloge locale pour diminuer la dérive.

Vous pouvez utiliser l'outil `ntpq` pour requêter l'état de NTP. En lançant :

```
ntpq -n
```

vous ouvrez un interpréteur interactif qui attend vos commandes. L'option `-n` empêche la résolution DNS des adresses de pairs, ce qui vous fait gagner un peu de temps.

Ensuite, la commande :

```
peers
```

vous montre la liste des machines qui sont requêtées par NTP. Voici le résultat sur notre machine :

```
remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
+37.187.107.140  193.190.230.66   2 u   23   64  377   39.847    1.850  56.700
+212.85.158.10   138.96.64.10     2 u    9   64  377   49.145    1.752  39.481
*37.187.101.146  192.93.2.20      2 u   39   64  377   40.056    4.875  35.558
+212.83.133.52   192.93.2.20      2 u   31   64  377   34.956    5.737  42.967
+91.189.89.199   131.188.3.220    2 u   39   64  377   82.707  -16.319  43.121
```

Il s'agit des machines définies dans le fichier ntp.conf. Voici la description de chaque colonne du tableau de résultat.

Tableau 3-6. Résultats de la commande peers

Colonne	Description
`Remote`	Adresse du pair ou du serveur qui est contacté.
`Refid`	Source de synchronisation de ce pair. Une machine à un stratum supérieur.
`St`	Stratum du pair. Ici, des machines au stratum 2.
`T`	Type de pair : u pour unicast, m pour multicast, l pour local.
`When`	Nombre de secondes depuis le dernier contact avec le pair.
`Poll`	Intervalle de requêtage vers ce pair en secondes.

Reach	Une valeur statistique d'accessibilité du pair, assez obscure à comprendre. Ici, notre valeur est le maximum : 377. Sur le binaire de 8 bits qui stocke cette valeur, cela correspond à 11111111 en octal (255 en décimal). Donc succès sur les derniers huit appels vers ce pair.
Delay	Délai en millisecondes pour obtenir la valeur du pair. Cette valeur doit naturellement être compensée par NTP pour la mise à jour de l'horloge locale avec la valeur récupérée du pair.
Offset	Différence de temps entre le pair et l'horloge locale, en millisecondes. C'est la valeur à appliquer à l'horloge locale.
jitter	Appelée aussi dispersion, différence maximale observée entre le pair et la machine. il s'agit d'une bonne indication de la dérive de l'horloge locale.

Vous pouvez aussi obtenir le statut de l'horloge locale avec la commande `rv` (pour `readvar` : lecture des variables).

Résultat chez nous :

```
associd=0 status=0618 leap_none, sync_ntp, 1 event, no_sys_peer,
version="ntpd 4.2.6p5@1.2349-o Wed Oct  9 19:08:06 UTC 2013 (1)",
processor="x86_64", system="Linux/3.13.0-24-generic", leap=00, stratum=3,
precision=-22, rootdelay=43.308, rootdisp=66.116, refid=37.187.107.140,
reftime=d8680b75.ae96db62  Tue, Jan 20 2015  0:02:13.681,
clock=d8680bc5.65b80e89  Tue, Jan 20 2015  0:03:33.397, peer=30832, tc=6,
mintc=3, offset=-11.594, frequency=-13.000, sys_jitter=6.558,
clk_jitter=6.278, clk_wander=1.598
```

Nous avons mis en gras les valeurs importantes. Ce résultat correspond à ce qui est défini dans la spécification RFC 1305 de NTP version 3. Vous voyez le stratum, le temps actuel de l'horloge, ainsi que le reftime qui représente la dernière fois où l'horloge a été synchronisée.

Vous pouvez ensuite quitter ntpq avec la commande `exit`.

Vector clocks

Si Cassandra utilise un horodatage simple, fourni par le client, pour gérer la cohérence des données, d'autres systèmes utilisent une méthode de gestion de versions interne, implémentée dans Dynamo de Amazon : la technique des *vector clocks* (tableaux d'horloges). C'est le cas notamment de Riak.

Un timestamp tel que celui utilisé dans Cassandra permet une gestion simple des versions, mais cette dernière ne maintient aucune information de causalité. Il n'est pas possible à partir d'une colonne et de ses différentes versions, de savoir si une version est descendante d'une autre version, directement ou indirectement, ou s'il s'agit d'une version parallèle. Cette information peut être utile pour une gestion de résolution de conflits plus sophistiquée. Le vector clock permet cette distinction. L'algorithme est simple : chaque version est marquée d'une information de vector clock composée d'un identifiant de nœud ou de processus (on pourrait dire, conceptuellement, un identifiant d'acteur), et d'un compteur de versions. Le tableau 3-7 reprend l'évolution du vector clock d'une donnée depuis l'insertion.

Tableau 3-7. **Évolution d'un vector clock**

Action	Valeur du vector clock
Nœud1 insère une donnée.	Nœud1:1
Nœud1 effectue une modification.	Nœud1:2
Nœud2 effectue une modification.	Nœud1:2, Nœud2:1
Nœud3 effectue une modification sur la version Nœud1:2.	Nœud1:2, Nœud3:1

Dans les deux dernières lignes du tableau, nous voyons que la version Nœud1:2 est modifiée par deux autres nœuds de façon concurrente. Grâce au vector clock il est possible d'en déduire qu'il y a un conflit, et de définir sur quelle version de la donnée ce conflit se produit.

Exemple d'utilisation dans Riak

Riak utilise les vector clocks pour sa gestion de cohérence et de conflits. Le vector clock est calculé par rapport à l'identifiant de la machine cliente qui envoie l'instruction. Nous allons l'expérimenter en utilisant son interface REST. Pour installer Riak et comprendre son fonctionnement, référez-vous au chapitre 9 consacré à Riak. Nous utilisons notre *bucket* (l'équivalent d'une base de données dans Riak) `Passerelles`, dont nous modifions les propriétés de la façon suivante :

```
curl -v -XPUT -H "Content-Type: application/json" -d '{"props":{"allow_mult":"true",
"last_write_wins":"false"}}' http://192.168.0.22:8098/buckets/passerelles/props
```

Nous avons indiqué les deux propriétés suivantes.

Tableau 3-8. **Propriétés du bucket**

Propriété	Description
`allow_mult`	Vrai = permet le stockage de plusieurs versions de la même donnée.
`last_write_wins`	Faux = conserve l'historique des vector clocks.

Nous vérifions que les propriétés ont été correctement enregistrées :

```
curl -i http://192.168.0.22:8098/buckets/passerelles/props
```

Voici une partie de l'information de retour :

```
{"props":{"allow_mult":true,"basic_quorum":false,"big_vclock":50,"chash_
keyfun":{"mod":"riak_core_util","fun":"chash_std_keyfun"},"dw":"quorum","last_write_
wins":false,"linkfun":{"mod":"riak_kv_wm_link_walker","fun":"mapreduce_linkfun"},"n_
val":3,"name":"passerelles","notfound_ok":true,"old_vclock":86400,"postcommit":[],
"pr":0,"precommit":[],"pw":0,"r":"quorum","rw":"quorum","sm
all_vclock":50,"w":"quorum","young_vclock":20}}
```

Ensuite, nous insérons un article de notre blog Passerelles :

```
curl -X PUT -H «X-Riak-ClientId: Client1» -H «Content-Type: application/json» -d '
{
«auteur»:
{
```

```
    «prénom»:»Annie»,
    «nom»:»Brizard»,
    «e-mail»:»annie.brizard@cocomail.com»
},
«titre»:»pourquoi les éléphants ont-ils de grandes oreilles?»
}' http://localhost:8098/buckets/passerelles/keys/1
```

En spécifiant une valeur manuellement dans l'en-tête `X-Riak-ClientId`, nous simulons l'envoi depuis un client spécifique, ici nommé `Client1`. Récupérons maintenant le document :

```
curl -i http://localhost:8098/buckets/passerelles/keys/1
```

Le retour inclut l'en-tête et le document. Nous reproduisons ci-dessous une partie de l'en-tête.

```
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVIcBckcawMWufJnMCUy5rEyxNwTO8WXBQA=
Vary: Accept-Encoding
Last-Modified: Tue, 13 Nov 2012 14:19:39 GMT
Date: Tue, 13 Nov 2012 14:19:43 GMT
Content-Type: application/json
Content-Length: 223
```

L'en-tête `X-Riak-Vclock` contient une version encodée du vector clock maintenu par Riak.

Nous mettons ensuite à jour :

```
curl -X PUT -H "X-Riak-ClientId: Client2" -H "X-Riak-Vclock:
a85hYGBgzGDKBVIcBckcawMWufJnMCUy5rEyxNwTO8WXBQA=" -H "Content-Type: application/json"
-d '
{
«auteur»:
{
    «prénom»:»Annie»,
    «nom»:»Brizard»,
    «e-mail»:»annie.brizard@cocomail.com»
},
«titre»:»Ce sont plutôt les lapins, non?»
}' http://localhost:8098/buckets/passerelles/keys/1
```

Nous avons envoyé le vector clock reçu dans l'en-tête, et nous avons indiqué un client différent : `Client2`.

Vector clock dans l'en-tête

Le vector clock doit être envoyé dans l'en-tête, sinon Riak considère qu'il s'agit d'une première version de la donnée, et la réécrit sans tenir compte d'éventuels conflits.

Essayons de créer un conflit en mettant de nouveau à jour en mentionnant toujours le premier vector clock dans l'en-tête, mais en spécifiant un client différent. Cela simule donc une mise à jour concurrente de la même version depuis deux clients différents.

```
curl -X PUT -H "X-Riak-ClientId: Client3" -H "X-Riak-Vclock:
a85hYGBgzGDKBVIcBckcawMWufJnMCUy5rEyxNwTO8WXBQA=" -H "Content-Type: application/json"
-d '
{
«auteur»:
{
    «prénom»:»Annie»,
    «nom»:»Brizard»,
    «e-mail»:»annie.brizard@cocomail.com»
},
«titre»:»Ce sont plutôt les lièvres, non?»
}' http://localhost:8098/buckets/passerelles/keys/1
```

En requêtant de nouveau notre article, voici ce que nous obtenons (sans l'en-tête) :

```
Siblings:
4dMeEvDd4OiOTRivwn17sP
79rzSSGm3FLXgiTp37Lak1
```

Il y a maintenant deux versions concurrentes. La dernière insertion n'étant pas un descendant de la précédente, Riak a conservé les deux versions et affiche un conflit.

Si nous relançons notre requête en indiquant dans l'en-tête HTTP au serveur que nous acceptons les résultats multiparts, de la façon suivante :

```
curl -i -H "Accept: multipart/mixed" http://192.168.0.22:8098/buckets/passerelles/
keys/1
```

nous obtenons le détail des versions :

```
HTTP/1.1 300 Multiple Choices
X-Riak-Vclock: a85hYGBgzGDKBVIcBckcawMWufJnMCUy57EyOOlKnuLLAgA=
Vary: Accept, Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted it blue)
Last-Modified: Thu, 15 Nov 2012 09:09:22 GMT
ETag: "2XIl3pbcyiyPslEjWAkMwf"
Date: Thu, 15 Nov 2012 09:18:03 GMT
Content-Type: multipart/mixed; boundary=VVvHewxojEOfURWT1OildkvVgim
Content-Length: 695


--VVvHewxojEOfURWT1OildkvVgim
Content-Type: application/json
Link: </buckets/passerelles>; rel="up"
Etag: 4dMeEvDd4OiOTRivwn17sP
Last-Modified: Thu, 15 Nov 2012 09:09:22 GMT

{
"auteur":
{
"prénom":"Annie",
```

```
"nom":"Brizard",
"e-mail":"annie.brizard@cocomail.com"
},
«titre»:»Ce sont plutôt les lièvres, non?»
}
--VVvHewxojEOfURWT1OildkvVgim
Content-Type: application/json
Link: </buckets/passerelles>; rel=»up»
Etag: 79rzSSGm3FLXgiTp37Lak1
Last-Modified: Thu, 15 Nov 2012 09:09:06 GMT

{
"auteur":
{
"prénom":"Annie",
"nom":"Brizard",
"e-mail":"annie.brizard@cocomail.com"
},
«titre»:»Ce sont plutôt les lapins, non?»
}
--VVvHewxojEOfURWT1OildkvVgim--
```

Pour résoudre le conflit, il suffira de mettre à jour la donnée en indiquant le vector clock renvoyé (ici `X-Riak-Vclock: a85hYGBgzGDKBVIcBckcawMWufJnMCUy57EyOOlKnuLLAgA=`) pour écraser les deux versions en conflit, en faisant un choix au niveau de l'application cliente.

Le Big Data analytique

Depuis le milieu de la décennie 2000, notamment après la parution des articles de Google dont nous avons déjà parlé, ce qu'on appelle maintenant le Big Data analytique a profité d'un fort développement, avec Hadoop comme fer de lance.

On peut classer les orientations Big Data dans les mêmes catégories que les bases de données traditionnelles: opérationnelles et analytiques, OLTP et OLAP. Une base de données opérationnelle vit et se trouve constamment modifiée pour soutenir une activité en perpétuel changement. Les moteurs NoSQL qui permettent de monter en charge sur des pétaoctets sont par exemple Cassandra ou HBase. Ils offrent des capacités de gestion et de stockage des données semblables aux SGBDR, à la différence près qu'ils sont distribuables sur des milliers de machines, utilisant des concepts comme les tables de hachage distribuées (pour Cassandra) ou la distribution par consensus (HBase).

Ce sont des outils optimisés pour les écritures et les lectures unitaires, non pas pour des traitements massifs des données et des agrégations. Pour cela, vous devez utiliser une approche analytique. La première de ces approches a été – et reste, c'est toujours la méthode principale – MapReduce, sous l'influence de Google.

Le paradigme MapReduce

MapReduce n'est pas en soi un élément de bases de données. Il s'agit d'une approche de traitement de l'information distribuée qui prend une liste en entrée et en produit une en retour. Il peut donc être utilisé pour de nombreux cas de figure et nous avons vu qu'il se marie bien avec les besoins de traitements (comme des calculs) distribués des bases NoSQL et les traitements décisionnels.

MapReduce a été défini en 2004 dans un article rédigé par Google. Le principe est simple : pour distribuer un traitement, Google a imaginé une opération en deux étapes. Tout d'abord, une attribution des opérations sur chaque machine (`Map`), suivie après traitement d'un rassemblement des résultats (`Reduce`). En soi, le raisonnement n'est pas révolutionnaire. Encore fallait-il en décrire l'implémentation technique, de façon suffisamment intelligente, pour éviter les écueils propres à un environnement distribué. Le traitement distribué génère des défis comme : que faire en cas de défaillance d'une unité de traitement ? comment s'assurer d'une bonne distribution du travail ? comment synchroniser de façon efficiente les résultats ? Le modèle MapReduce offre des réponses à ces défis avec la volonté de simplifier et de contourner les problèmes pour aboutir à des solutions pragmatiques.

Les besoins de Google qui ont donné naissance à MapReduce sont de deux ordres : comment traiter des volumes gigantesques de données déstructurées (des pages web à analyser pour nourrir le moteur de recherche de Google, ou l'analyse des logs produits par le travail de ses moteurs d'indexation, par exemple), pour en tirer des résultats de calculs, des agrégats, des résumés… bref, de l'analyse.

La diminution du coût du matériel ouvre la voie à la résolution du problème de l'analyse du Big Data. Créer une architecture logicielle qui permette une montée en charge horizontale simple est la réponse la moins coûteuse au problème de la montée en charge. Optimiser les performances d'un système de gestion de données sur une seule machine demande beaucoup d'énergie et de compétences, pour aboutir à des résultats fragiles qui ne peuvent supporter une multiplication soudaine de la demande. En revanche, s'assurer que le modèle de traitement des données est bien distribué de la façon la plus élégante possible sur des machines séparées, qui peuvent être multipliées à l'infini, permet de répondre à des augmentations foudroyantes de la demande de façon très simple : par l'achat et l'installation rapide de nouvelles machines, qu'elles soient puissantes ou non, et en s'assurant que toute défaillance d'une machine ne se traduise pas par une perte de données. Il faut donc vérifier que le modèle déployé est capable de distribuer au mieux les données et le travail, même sur des dizaines de milliers de nœuds, en offrant un système de réplication suffisant pour éliminer statistiquement les risques de pertes de données.

Ce sont ces critères que le modèle MapReduce permet de respecter. Il offre un modèle qui simplifie au minimum les complexités de l'informatique distribuée, en vue de fournir des outils aux développeurs. L'implémentation en Java donne par ailleurs la possibilité de s'abstraire de l'architecture matérielle, afin qu'un framework MapReduce puisse fonctionner et interagir sur des machines hétérogènes.

Les opérations de `map` et `reduce` imaginées par Google ont été inspirées par les primitives du même nom du langage Lisp. Il est ici important de comprendre le modèle de programmation qui a donné naissance à MapReduce.

Lisp et les langages fonctionnels

Lisp est l'un des plus anciens langages informatiques. Il est cependant encore apprécié de nombreux développeurs. Lisp se base sur un modèle de programmation très différent des langages impératifs dont nous avons parlé pour les opposer au langage déclaratif qu'est SQL. L'ordinateur et les langages impératifs sont basés sur le concept de machine de Turing, où le traitement des données produit des changements d'état. À un instant donné, la machine a un état, qui correspond à l'ensemble des valeurs qui y sont placées. Un langage manipule cet état et permet de le faire évoluer à travers un certain nombre d'actions. Un langage impératif manipule et modifie cet état, en affectant des valeurs à des variables, par exemple. Le problème de cette approche est qu'un langage affectant en permanence l'état du système peut produire des effets de bord sur d'autres sous-ensembles de ce système. Par exemple, si vous vivez en famille et que les membres de votre famille qui remplissent le frigo ne sont pas toujours les mêmes que ceux qui y puisent leur nourriture, les effets de bord sont évidents : Élise fait des courses et achète deux yaourts pour les consommer dans quelques jours, mais entre-temps, William qui ouvre le frigo au moment où il a un petit creux, mange d'un coup les deux yaourts. Lorsque Élise a faim à son tour, elle ne trouve plus de yaourts dans le frigo et ne peut se sustenter. L'état du frigo peut être changé par plusieurs acteurs, il est donc en quelque sorte imprévisible pour chacun d'entre eux, et cela produit des effets de bord. On n'est jamais sûr de retrouver le frigo dans l'état où on l'a laissé la dernière fois qu'on l'a ouvert.

La programmation fonctionnelle répond à ce problème en proposant un modèle différent. L'état du système global ne sera jamais altéré. Les routines d'un langage fonctionnel ne modifient pas l'état général du système, mais elles ne maintiennent qu'un état transitoire, propre à chaque routine, et retournent des résultats qui seront toujours manipulés de façon locale. Pour cela, le modèle principal de la programmation fonctionnelle consiste à utiliser des fonctions.

La fonction

En mathématiques, une fonction utilise une variable et des paramètres constants pour retourner une valeur. Un appel de fonction effectue donc un calcul dont le résultat est retourné à l'appelant de la fonction.

L'application systématique de fonctions permet de manipuler des données sans maintenir un état général, car aucun état n'est maintenu hors du temps d'exécution de la fonction. Pour cela, les mécanismes d'exécution de ces fonctions sont privilégiés dans les langages fonctionnels. Par exemple, l'appel récursif des fonctions remplace la programmation par boucle, ce qui permet de s'affranchir de l'utilisation de variables mutables (auxquelles on peut affecter une nouvelle valeur). La programmation fonctionnelle offre la possibilité de gérer de façon très souple et sûre le flux de la donnée, car aucun état hors de l'exécution n'est maintenu. Cela permet donc de réduire les effets de bord et de gagner en souplesse au niveau du flux du programme, en acceptant par exemple plusieurs ordres d'exécution.

Le langage Lisp est né dans les années 1950 et se reconnaît facilement par son nombre élevé de parenthèses (au point que son acronyme est parfois transformé en *Lost In Silly Parenthesis*). Lisp manipule des listes, auxquelles des fonctions sont appliquées. À l'extrême, Scheme, un langage fonctionnel dérivé de Lisp, considère tout comme étant une s-expression, c'est-à-dire que Scheme

ne fait aucune différence entre du code et des données. Cela permet la création de fonctions d'ordre élevé, c'est-à-dire des fonctions qui acceptent d'autres fonctions comme paramètres.

Un exemple de MapReduce en Lisp

Comme Lisp ne manipule que les listes, il a besoin d'un mécanisme pour appliquer une fonction à chaque membre d'une liste. Cette fonction s'appelle `Map`. Elle retourne une nouvelle liste dont chaque membre est modifié. Par exemple, le code suivant peut être exécuté dans le célèbre éditeur Emacs, qui est presque entièrement écrit en Lisp :

```
(require 'cl)
(map #'list '1+ '(1 2 3 4))
```

Tout d'abord, nous ajoutons le module `cl` qui permet d'interpréter des instructions écrites en Common Lisp au lieu de Elisp (Emacs Lisp) propre à Emacs. Nous utilisons ici Emacs comme interpréteur Lisp car il est libre, facile à obtenir et à installer sur tous les systèmes. L'exécution de cette ligne dans Gnu Emacs (à l'aide de la combinaison de touches Ctrl + Alt + X) retourne :

```
(2 3 4 5)
```

Nous avons appliqué l'opération 1+ à chaque membre de la liste, et nous recevons en résultat une nouvelle liste dont chaque membre est transformé.

La fonction `Reduce`, aussi appelée `Fold` dans d'autres langages fonctionnels comme Haskell, permet d'effectuer un traitement sur chaque élément de la liste pour retourner une seule valeur. `Reduce` utilise en interne un principe itératif : un accumulateur est d'abord initialisé. La fonction `Reduce` est appliquée à chaque élément de la liste et l'accumulateur est incrémenté. Le résultat est la valeur de l'accumulateur après traitement de chaque élément de la liste. La ligne de code suivante, exécutée dans Emacs (après avoir ajouté le module `cl` comme précédemment) :

```
(reduce '+ '(2 3 4 5))
```

retourne simplement 14.

Ces deux fonctions ont donc donné des idées aux ingénieurs de Google. Pourquoi ne pas appliquer les mêmes principes de façon distribuée ? Des fonctions `Map` sont appliquées sur un grand nombre de machines, générant comme résultat des paires clé-valeur. Ces ensembles de paires clé-valeur sont ensuite regroupés et une fonction `Reduce` s'applique à combiner les valeurs qui appartiennent à des clés identiques, pour retourner une valeur agrégée par clé. C'est un modèle qui peut s'appliquer à un grand nombre de problèmes.

Ce sont ces principes qui ont conduit à l'élaboration de MapReduce. Un algorithme de type Map traite une liste élément par élément, de façon isolée. Cela rend donc possible une exécution en parallèle et même une exécution distribuée sur plusieurs nœuds.

Le fonctionnement d'Hadoop MapReduce

L'implémentation libre de référence de MapReduce s'appelle Hadoop. Nous en avons parlé : développement par une équipe menée par Doug Cutting, en Java, pour les besoins de son moteur

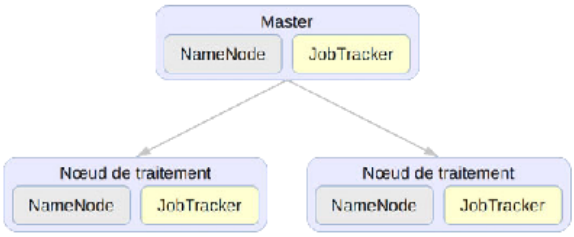
d'indexation distribué Nutch pour Yahoo!. Hadoop implémente directement le document de Google sur MapReduce, et base son stockage distribué sur HDFS, qui implémente le document de Google sur GFS. Résumons : Hadoop est un framework destiné à distribuer du traitement de données sur un nombre potentiellement illimité de nœuds.

Il est beaucoup plus efficace d'amener le traitement aux données que l'inverse. C'est la même idée qui est à l'œuvre dans les SGBDR : pour obtenir de bonnes performances avec un moteur comme Oracle ou SQL Server, vous créez des procédures stockées qui exécutent le code de traitement au plus proche des données. Si vous récupérez un large jeu de résultat pour le traiter sur votre client, les performances s'effondrent.

Hadoop MapReduce vous permet d'écrire des fonctions Map et Reduce en Java, et de les envoyer sur des nœuds de traitement. Ces machines comportent aussi un gestionnaire de données (HDFS, HBase, Cassandra…) que MapReduce va interroger pour récupérer ses données. Pour que MapReduce fonctionne, il faut que le traitement et les données soient colocalisées : on parle de localité des données. Le traitement s'effectuant sur les données locales à chaque nœud, le traitement sur les données est donc aussi efficace que celui d'une procédure stockée dans un SGBDR : on traite au plus proche des données.

Nous allons parler encore un instant de cette architecture, à l'aide du schéma très simple de la figure 3-13.

Figure 3-13
Architecture d'Hadoop MapReduce



Imaginons un traitement MapReduce avec des données stockées sur HDFS. Hadoop MapReduce est basé sur une architecture maître-secondaire. Le maître, qui exécute un processus nommé le JobTracker, a la responsabilité de gérer l'exécution d'une opération MapReduce. HDFS est lui aussi basé sur une architecture maître-secondaire. Le maître s'appelle le NameNode. C'est la table des matières qui contient l'arbre des répertoires et des fichiers du système de fichiers distribué. C'est donc lui qui sait où sont les données. Le JobTracker contacte le NameNode pour savoir où se trouvent les données qu'il aura à traiter.

Les nœuds secondaires MapReduce comportent des processus TaskTracker. Le JobTracker envoie le code de traitement aux TaskTrackers libres qui sont au plus proche des données, idéalement en localité.

La responsabilité du JobTracker est de superviser le travail des TaskTrackers. Ceux-ci vont renvoyer régulièrement un signal (un heartbeat) au JobTracker. Tout TaskTracker qui ne se manifeste pas dans un délai défini est considéré comme perdu, et son travail est attribué à un autre

TaskTracker. De même, le TaskTracker peut renvoyer une notification d'erreur, et le JobTracker peut choisir d'exécuter le code ailleurs et de marquer les données ou le TaskTracker comme mauvais. Le JobTracker marquera à la fin son statut comme terminé, ce que le client pourra savoir en interrogeant régulièrement le JobTracker.

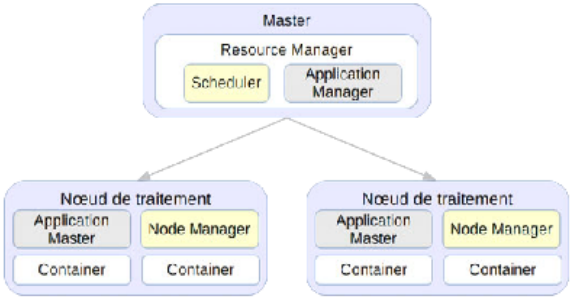
Hadoop 2 YARN

Ce que nous venons de décrire, c'est un framework qui offre des fonctionnalités de distribution d'un traitement, Hadoop, et un type de traitement spécifique, MapReduce. Tout le travail vers Hadoop 2, nom de code YARN (*Yet Another Resource Negotiator*), a été de séparer ces deux élements.

Pour la sortie de la version 0.23 d'Hadoop, une grande réorganisation a été effectuée. La nouvelle version s'appelle MapReduce 2.0 (MRv2) ou YARN. Le Jobtracker a été séparé en deux : un module de gestion de ressources (*Resource Manager*, ou RM) et un module de traitement appelé l'*Application Master*, ou AM.

Le Resource Manager coordonne le travail entre les nœuds, sur lequels un démon client, nommé le Node Manager, gère l'exécution locale du travail. L'exécution des fonctions MapReduce est déléguée à un démon local séparé, l'Application Master, qui utilise le Resource Manager pour l'exécution et la planification du travail.

Figure 3-14
Architecture d'Hadoop YARN



Le Resource Manager comporte lui-même deux modules : d'abord, un *scheduler* responsable de l'ordonnancement des tâches, dans le sens où il va simplement allouer des ressources pour l'exécution des tâches. C'est un ordonnanceur générique pluggable. On peut donc lui attribuer dynamiquement des algorithmes spécifiques d'ordonnancement.

Un exemple : l'ordonnanceur *FairScheduler* est un module dont l'objectif est d'assurer une distribution équitable des ressources sur le cluster dans le temps, en créant des files de traitement. Une description du FairScheduler est disponible à cette adresse : http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html. Pour l'instant, il n'y a que deux modules : FairScheduler et CapacityScheduler (http://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html).

Le scheduler est complètement agnostique vis-à-vis du type de traitement qu'il va prendre en charge, tout comme l'ordonnanceur d'un système d'exploitation ne sait rien de la nature des processus qu'il place sur des processeurs. De plus, il ne s'occupe que de l'allocation des ressources. La gestion du replacement des tâches en cas d'erreur n'est pas de son ressort.

Les ressources en question sont pour l'instant seulement de la mémoire vive. Cela va évoluer en incluant les autres types de ressources d'une machine. Le Resource Manager définit des containers, qui sont des unités de ressources présents sur un nœud. La distribution du traitement va s'opérer sur des containers sur des nœuds.

Ensuite, le deuxième module du Resource Manager est l'Application Manager. C'est un ordonnanceur de tâches qui est en contact avec le code de traitement proprement dit. L'Application Manager reçoit la demande du client et lance l'Application Master. Ce dernier est le gestionnaire du traitement spécifique, par exemple un travail MapReduce. C'est lui qui va distribuer le traitement sur les containers mis à disposition par le Resource Manager.

Séparation des tâches

Cette séparation entre deux modules, le RM et l'AM, signifie donc que Hadoop YARN (le RM) est un pur ordonnanceur de tâches distribué qui est agnostique en matière de traitement. Il permet de lancer n'importe quel type de tâche, et non plus seulement du MapReduce, qui devient un type spécifique d'AM. Vous pouvez donc distribuer d'autres types de traitement que du MapReduce. Ces traitements prennent un terme plus générique de DAG, ou *Directed Acyclic Graph* (graphe orienté acyclique). Ce terme un peu compliqué, qui emprunte à la théorie des graphes, décrit de façon générique un concept simple: des nœuds reliés entre eux sans qu'ils forment un circuit. Un DAG dans le sens YARN, c'est un traitement distribué qui peut contenir tout type de liaison entre les parties du traitement. La liaison entre un traitement Map et un traitement Reduce est une forme de DAG.

Mesos, le concurrent

YARN n'est pas le seul environnement de traitement disponible. Sous la bannière Apache, vous trouvez aussi Mesos, un gestionnaire de cluster développé à l'université de Californie à Berkeley, qui reprend conceptuellement l'architecture du noyau Linux pour l'abstraire sur un cluster de machines. Il offre donc un environnement sur lequel tourner du traitement distribué. C'est un projet très actif et très prometteur, qui s'avère un sérieux concurrent de YARN.

Hadoop 2 libère Hadoop de MapReduce et permet de bâtir Hadoop YARN et HDFS comme un environnement de choix pour la distribution de tous types d'analyses de données Big Data.

On assiste aujourd'hui à une floraison d'applications qui se basent sur YARN, la plupart donnés à la fondation Apache. Vous avez, avec ces applications, tout ce qu'il vous faut pour échanger et traiter vos pétaoctets de données de façon distribuée, en batch ou en temps réel.

Mode batch *vs* temps réel

MapReduce, c'est un algorithme qui fonctionne en mode batch : il n'y a pas de possibilités d'interrogation interactive des données, ni de traitement immédiat de l'information. Typiquement, comme pour l'exemple canonique de l'indexation d'un moteur de recherche comme Google, vous lancez un

traitement planifié, qui prend le temps qu'il doit prendre, et qui écrit son résultat dans un fichier, conceptuellement une vue matérialisée du résultat de l'analyse des données. Cela peut prendre quelques heures. Il existe un concours annuel de performance d'exécution d'un traitement de tri sur un cluster, dont vous trouvez les résultats sur le site http://sortbenchmark.org/. Par exemple, en 2013, le cluster Hadoop de Yahoo! est parvenu à trier 102,5 TO en 4 328 secondes sur un cluster de 2 100 nœuds, c'est-à-dire en une heure et douze minutes. Plutôt pas mal, cela représente une capacité de traitement de 1,42 TO par seconde… Regardez les résultats 2014, Apache Spark va au moins trois fois plus vite sur 207 nœuds Amazon EC2. Nous reparlerons de Spark.

Mais, aussi impressionnant que soient ces performances, elles ne permettent pas de répondre à des besoins comme un traitement en temps réel des informations: réaction automatique à des situations émergentes, comme les tendances sur les réseaux sociaux, des conditions actuelles de trafic autoroutier, des recommandations à l'utilisateur basées sur son activité du moment, etc. En fait, au fur et à mesure que le Big Data se démocratise, ces besoins se font de plus en plus pressants, et les exemples abondent. Du côté analytique, cela ne permet pas à l'utilisateur de miner ou d'interroger ses données de façon interactive, en exécutant par exemple un langage d'interrogation ad hoc, comme HiveQL, et d'obtenir rapidement sa réponse, pour la modifier au besoin et renvoyer sa requête. Ces deux besoins commencent maintenant à être couverts par des applications spécifiques qui utilisent YARN ou Mesos comme couche de distribution. Par exemple, Apache Spark offre des possibilités de requêtage interactif et de traitement de flux en temps réel, et Apache Storm est un système de traitement de flux en temps réel à la popularité croissante. Nous étudierons plus en détail ces outils dans la partie II du livre, lors de notre exploration des différentes solutions NoSQL.

Le Big Data interactif

On pourrait établir une loi pragmatique: pour savoir ce qui va se passer en Big Data, surveillez les publications de Google (http://research.google.com/pubs/papers.html). En 2010, Google publie un article qui, une fois de plus, décrit un système développé en interne. Il s'intitule «Dremel: Interactive Analysis of Web-Scale Datasets» (Dremel, analyse interactive d'ensembles de données de taille web). Ce système garantit l'analyse interactive de très grands volumes en quelques secondes sur un cluster de traitement.

Dremel est basé d'abord sur un stockage spécifique des données dans GFS. La structure est définie comme un stockage orienté colonnes pour des données imbriquées *(columnar storage format for nested data)*. Le modèle de données, d'un point de vue logique, est celui de Protocol Buffers (voir la section «Les protocoles d'accès aux données» en début de ce chapitre). Il s'agit donc d'une structure typée, ordonnée et hiérarchique. Des éléments peuvent être marqués comme obligatoires, optionnels ou répétés. On peut faire référence à une donnée dans une structure protobuf à l'aide d'un chemin, comme on peut le faire en XML ou en JSON.

Reprenons ici la structure que nous avions proposée dans la section décrivant Protobuf page 59:

```
message Article {
  required int32 id = 1;
  required string titre = 2;
```

```
  message Auteur {
    required string prenom = 1;
    required string nom = 2;
    required string email = 3;
  }

  repeated Auteur auteur = 3;
}

message Articles {
  repeated Article article = 1;
}
```

Et créons un document de cette structure :

```
id: 1;
titre: 'Faut-il vraiment prendre des vacances?'
auteur:
       prenom: 'Antoine'
    nom: 'Schmidt'
    email: 'aschmidt@passerelles.fr'
auteur:
       prenom: 'Pascale'
    nom: 'Chemin'
    email: 'pchemin@passerelles.fr'
```

Le chemin d'un e-mail d'auteur, par exemple, peut donc être exprimé ainsi : `auteur.email`. Chaque chemin est stocké séparément dans des tables, de façon à pouvoir y accéder indépendamment du reste du document. Un algorithme permet de reconstruire le document à partir des différents stockages parce que des informations de répétition et de position sont ajoutées dans les tables. Grace à ce type de stockage, il est possible de reconstruire efficacement une partie du document sans accéder au document tout entier, pour les besoins de l'analyse.

Les données sont stockées dans GFS ou BigTable ; le langage d'interrogation est un dérivé de SQL, qui travaille avec ce concept de tables imbriquées, et qui retourne en résultat de nouvelles tables imbriquées, c'est-à-dire des structures définies en protobuf. La requête descend un arbre d'exécution, lequel passe d'abord par une machine racine identifiant les machines qui vont contenir les données à requêter, qui envoie la requête sur des serveurs intermédiaires, qui envoient à leur tour cette requête, etc., jusqu'à des machines au dernier niveau, au plus proche des données, qui font le travail. Un module nommé le *Query Dispatcher* distribue les différentes requêtes et s'occupe de la tolérance de panne, un peu à la manière du JobTracker d'Hadoop.

La fin du document de Google indique des statistiques de performances très impressionnantes, jusqu'à pouvoir parcourir 100 milliards de lignes par seconde.

Suivant l'exemple des précédentes publications de Google, cet article a montré à la communauté que l'interrogation interactive des données précédemment traitées en mode batch par Hadoop était possible. Il ne restait plus qu'à commencer à y travailler. L'une des réalisations qui en a résulté s'appelle Impala, développé par Cloudera, une société bâtie autour d'Hadoop. Impala utilise un moteur d'exécution de requête conçu selon les principes des moteurs relationnels, avec la

possibilité d'exprimer des jointures entre jeux de données. Une autre implémentation libre toute fraîche s'appelle Apache Drill (http://drill.apache.org/), sortie durant l'été 2014. Dans la vraie vie, le Dremel est un moteur rotatif à haute puissance, d'où le nom de Drill qui reprend cette idée. Drill représente en interne les données provenant de différentes sources sous forme de documents proches du JSON, ce qui se rapproche de la stucture protobuf dont nous venons de parler. Un moteur de requête semblable à celui de Dremel offre d'excellentes performances et des connecteurs JDBC et ODBC permettent d'utiliser Drill à partir d'un langage client. C'est donc dans ce domaine que les avancées sont les plus prometteuses à l'heure actuelle.

4

Les schémas de données dans les bases NoSQL

Comme nous l'avons vu, le modèle relationnel repose sur une stabilité du schéma des données, et donc sur une modélisation poussée, idéalement complète, effectuée au préalable. Il est malheureusement difficile de respecter cette contrainte à la perfection, et nous ne nous souvenons pas avoir jamais rencontré un système relationnel dans lequel le schéma était resté invariant depuis sa première mise en production. Les bases NoSQL cherchent à assouplir cette contrainte, généralement en proposant une approche dite « sans schéma », ou à schéma « relaxé », ce qui veut dire qu'aucune vérification ou contrainte de schéma n'est effectuée par le moteur. C'est au développeur qui utilise ces systèmes de décider comment il organise ses données, et ceci dans son code client.

Le schéma implicite

Dans les faits, il est quand même difficile d'offrir une totale liberté. Par exemple, MongoDB se déclare schema-less (sans schéma), et en effet, rien n'empêche l'utilisateur de stocker des documents de différentes structures dans la même collection. Toutefois, ces documents ont un format structuré (appelé BSON), et il est fortement recommandé de conserver une structure homogène dans la même collection, pour des raisons de logique et notamment d'indexation. Tout mélanger n'aurait aucun sens, comment effectuer une recherche avec des critères communs ? Sur quoi placer un index ?

Le terme « schema-less » peut faire penser qu'un moteur NoSQL n'a pas de schéma du tout. Les moteurs purement paires clé-valeur peuvent prétendre à cette appellation : ils stockent une clé non

typée et une valeur opaque, purement binaire. On y met ce qu'on veut. En revanche, les moteurs orientés documents et orientés colonnes maintiennent une structure des données. Simplement, il n'y a aucun contrôle ou validation de cette structure côté serveur. On parlera donc plutôt de schéma implicite. Dans ce modèle, la responsabilité de s'assurer que le schéma est correct est déportée dans le code client. La responsabilité de la qualité des données incombe donc au développeur, et non plus à un DBA de moteur relationnel.

Schema-less est un terme plus militant que descriptif. Toute donnée complexe manipulée en programmation est une structure, elle a besoin d'un schéma qui décrit cette structure. Ce qu'on stocke dans un SGBD a donc un schéma. La différence est que, contrairement aux moteurs relationnels où le schéma est explicite (contrôlé par le serveur) et fortement structuré, les moteurs NoSQL possèdent un schéma implicite et semi-structuré.

Il est probable qu'avec le temps, les grands moteurs NoSQL intégreront des fonctionnalités de validation de schéma. C'est déjà le cas de Cassandra 2, où la structure des tables doit être définie à l'avance quand on utilise CQL 3. En ce qui concerne les documents, il existe une initiative pour développer une validation du JSON: http://json-schema.org/.

Pour le modèle relationnel, les données sont importantes, et leur qualité ainsi que leur organisation sont essentielles. Elles sont structurées, contraintes, protégées et cohérentes. En stockant vos données dans une base NoSQL, vous leur accordez un écrin différent, plus permissif. Vous êtes en quelque sorte comme un parent cool qui laisse la chambre de ses enfants dans l'ordre qu'ils souhaitent. Dans le monde de la base de données, les enfants sont les développeurs, et les données représentent la chambre.

Une approche non relationnelle

Par définition, le mouvement NoSQL, qui devrait, comme nous l'avons vu, s'appeler plutôt le mouvement non relationnel, s'échappe de ce qui fait le fondement des SGBDR : les relations entre des données structurées en tables, et donc les jointures nécessaires dans le code de requête pour rejoindre les données selon leurs relations. Si vous êtes familier avec le modèle relationnel, vous vous demandez peut-être comment faire pour stocker les données hautement répétitives dans un moteur NoSQL, par exemple les informations de référence comme les lieux, les catégories, etc.

En fait, rien n'empêche de bâtir ses bases NoSQL selon au moins quelques principes relationnels. Une base de données orientée documents, par exemple, référence un document par une clé unique. Si vous considérez une base CouchDB ou MongoDB qui contient un catalogue de DVD et une liste de commandes sur ces articles, le document détaillant la commande pourrait très bien contenir la clé de la commande, comme illustré sur la figure 4-1.

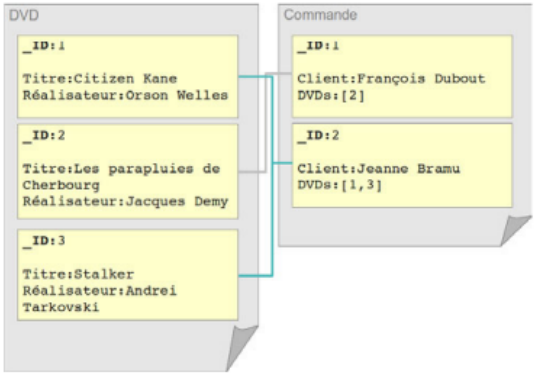


Figure 4-1
Relations dans une base de données documents

Vous viendrait-il à l'idée de stocker à nouveau dans chaque commande les informations complètes des DVD achetés ? À cette question, la documentation de MongoDB répond de la façon suivante : il y a deux manières de considérer la situation. Soit vous faites de l'enchâssement *(embedding)*, c'est-à-dire que vous intégrez des sous-documents dans le document BSON, et donc vous dupliquez vos données, soit vous faites des liaisons *(linking)*, et dans ce cas, vous devez gérer le déréférencement de ces liaisons dans votre programme client. Le raisonnement est qu'une gestion de relations au niveau du serveur serait difficile à effectuer dans un environnement distribué. MongoDB Inc. est donc clair à ce sujet : la voie conseillée est d'enchâsser les documents. Est-ce toujours la chose à faire ? Sans doute, la réponse se trouve toujours au même endroit : comment allez-vous utiliser vos données ? L'enchâssement des documents peut être intéressant si vous voulez conserver un historique des produits. La duplication permet de conserver à tout jamais l'état du DVD au moment où la commande a été passée. De plus, il n'est pas beaucoup plus difficile de rechercher toutes les commandes du même DVD avec un document enchâssé ou lié, si l'identifiant du DVD est maintenu de façon cohérente.

Les paires clé-valeur

Beaucoup de bases NoSQL partent du principe qu'une paire clé-valeur permet de répondre à la quasi-totalité des besoins. Cette paire clé-valeur ressemble d'ailleurs à ce qu'on peut connaître dans une table d'un schéma relationnel si on le simplifie. Une table dans une base SQL est composée d'une clé primaire et d'une ligne qu'elle identifie, ce qui pourrait être considéré comme une valeur.

Notion de clés candidates

Cette comparaison, nous sommes forcés de le reconnaître, est un peu faussée, parce qu'elle ne prend pas en compte la plus grande complexité du modèle relationnel. Une table peut comporter plusieurs clés candidates, ce qui permet d'identifier une ligne à partir de plusieurs clés, qu'elles soient appliquées sur une colonne ou sur un ensemble de colonnes.

Le raisonnement de la paire clé-valeur est que les besoins d'accès aux données peuvent être simplifiés ainsi : à partir d'une clé de recherche, je veux obtenir une donnée complexe. Voici quelques exemples d'application de cette règle.

- Sur un site de réseau social, à partir d'un utilisateur (la clé), je veux obtenir une liste de ses amis (la valeur).
- Dans un catalogue de livres, le numéro ISBN (la clé) donne accès à tous les détails sur le livre (la valeur).
- Dans un journal d'activités, la date d'un événement (la clé) indexe les détails de ce qui s'est passé à ce moment (la valeur).

Comme vous le voyez, la clé représente une information précise et atomique, alors que la valeur peut être complexe et représenter un tableau ou une liste, elle-même constituée de clés qui pointent sur des sous-valeurs, et ainsi de suite.

Il s'agit en fait d'un modèle bien connu, qu'on appelle souvent tableau associatif ou dictionnaire dans les langages de programmation. Par exemple, en Python, un dictionnaire se définit de la façon suivante :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

article = {} # ou dict()
print type(article)

article["auteur"] = {"prénom":"Annie", "nom":"Brizard", "e-mail":"annie.brizard@
cocomail.com"}
article["titre"] = "pourquoi les éléphants ont-ils de grandes oreilles?"

print article
```

Dans la plupart des pilotes de moteurs NoSQL développés pour les langages clients, les méthodes acceptent les structures de données du langage et les convertissent automatiquement en structures du moteur, en protobuf ou Thrift par exemple.

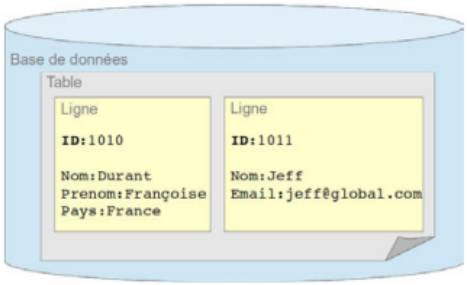
Les entrepôts clé-valeur

On peut opposer les entrepôts, qui stockent spécifiquement des paires clé-valeur, aux bases orientées documents, dont les valeurs sont des structures complexes comme des documents JSON. Tous deux ont des clés, mais les valeurs contenues dans les entrepôts clé-valeur sont considérées

comme scalaires. Ces entrepôts ont en général comme objectif d'offrir les meilleures performances possibles à travers un dictionnaire.

Un des objectifs des entrepôts clé-valeur est la simplicité. Il n'y a vraiment plus de notion de schéma de données dans un tel système. Chaque ligne peut être complètement différente des autres, et s'il n'y a pas de système d'indexation secondaire, le seul accès valable est par la clé de la ligne.

Figure 4-2
Entrepôt clé-valeur



Les moteurs spécifiquement clé-valeur sont des clones de Dynamo, comme Projet Voldemort ou Riak, et des descendants de memcached comme Redis (bien que Redis offre un modèle plus riche, voir chapitre 10).

On peut considérer que tous les moteurs NoSQL sont des entrepôts de paires clé-valeur plus ou moins spécialisés, et que les autres types de données que nous abordons dans ce chapitre dérivent du concept de clé-valeur. De même, ces moteurs évoluent souvent vers un schéma document de type JSON, en supportant l'indexation secondaire. Riak et Couchbase Server en sont des exemples.

La question de la redondance

Les bases NoSQL ne sont pas économiques en espace disque : on peut le comprendre du fait qu'il n'y ait pas de relation entre les tables, et que les documents contiennent souvent l'intégralité des données à travailler. Une base NoSQL sera susceptible de peser beaucoup plus lourd que les mêmes données stockées dans une base relationnelle bien normalisée. Cela ne gêne pas les développeurs et utilisateurs des bases NoSQL car l'absence de jointures leur paraît un immense pas en avant vers les performances et la simplicité (beaucoup de développeurs n'aiment pas les jointures, probablement des utilisateurs de MySQL, comme nous l'avons vu). De plus, le stockage de masse étant relativement accessible, les tailles importantes de données ne sont plus gênantes, en tout cas sur le papier.

Le problème des performances en NoSQL diffère radicalement de ce qu'il est dans un SGBDR. Un moteur relationnel tente d'offrir les meilleures performances possibles dans son contexte, en assurant une intégrité des données et une cohérence transactionnelle. Le point de focalisation des performances dans un moteur relationnel se situe au niveau des entrées-sorties. Manipuler des données ne change pas tant les CPU, mais principalement le système de stockage. C'est autour de ce problème que ce sont construits les moteurs relationnels, en offrant un modèle qui permet

d'éviter toute redondance de données, en maintenant un cache des données en mémoire (le *buffer pool*), et parfois en offrant des fonctionnalités de compression à la volée des pages de données qui, dans la plupart des cas, améliorent sensiblement les performances car le gain en I/O dépasse nettement l'augmentation de l'activité CPU.

Les moteurs NoSQL répondent à cette problématique de façon très différente. L'absence de relation induit une surface des données beaucoup plus grande, car les informations sont stockées de façon redondante. Il est bien sûr possible de simuler des relations entre les documents. Par exemple, on peut référencer la clé d'une paire clé-valeur dans la valeur d'un autre document. Imaginons que nous devions gérer un réseau social. Un document peut contenir les informations d'un utilisateur, avec sa clé (que ce soit son login ou une clé numérique) et la valeur sera un document JSON stockant des informations sur ses choix, ses activités et sa liste d'amis. Cette liste d'amis sera bien sûr une liste de clés d'autres utilisateurs. Pour créer la liste d'amis, le code client devra retrouver le document de l'utilisateur, puis les documents des amis à partir des clés. On pourrait considérer cela comme une jointure qui s'ignore, et il n'y a finalement pas beaucoup de différence avec un moteur relationnel, si ce n'est que ce dernier effectue cette jointure au niveau du serveur en utilisant un algorithme de jointure optimisé, alors que le système NoSQL va probablement effectuer ces opérations en boucle, peut-être avec des allers-retours du client au serveur.

En réalité, si on s'en tient purement aux moteurs eux-mêmes, les systèmes NoSQL sont plus lents, parfois beaucoup plus lents, que les systèmes relationnels, en fonction du type de recherche (excepté les moteurs en mémoire comme Redis). Ils sont plus lents en extraction, et parfois en modification. Paradoxalement, ils peuvent donner une impression de plus grande rapidité parce qu'ils distribuent la charge sur de nombreuses machines ou qu'ils dupliquent les données. C'est un peu la victoire des muscles sur l'intelligence.

SGBD ou entrepôt de données ?

Ce n'est peut-être pas très charitable de comparer les moteurs relationnels et NoSQL sur ces principes, même si ce sont eux qui ont commencé en se nommant « NoSQL ». Des outils comme Redis ou Riak ne se présentent pas comme des moteurs de bases de données mais comme des dépôts *(store)*. Le focus n'est pas sur la manipulation et la richesse fonctionnelle, ou sur les performances pures du moteur, mais sur le stockage et sur l'utilisation en RAM et la distribution de la charge de travail.

La clé et les recherches

L'organisation des données en paires clé-valeur, la valeur étant scalaire ou un document, pose aussi des problèmes de performances si l'on se place du point de vue de la base relationnelle. Si le point d'entrée est une clé, cela signifie que la seule recherche efficace passe par cette clé, car dans un moteur NoSQL, par défaut, seule la clé est indexée. Dans notre exemple de réseau social, nous pourrons rapidement accéder aux données globales d'un utilisateur grâce à la clé.

Mais qu'en est-il des recherches à partir d'autres critères ? Dans un moteur relationnel, il faudra soit effectuer un parcours (un *scan*), soit utiliser un index. Plusieurs moteurs NoSQL comme MongoDB ou Riak 2i permettent de créer des index, souvent appelés index secondaires, qu'ils soient en nouvelles paires clé-valeur ou en B-Tree. En ce qui concerne la recherche par scan, qui consiste à parcourir toutes les données pour tester la valeur à rechercher, elle n'est en général pas

implémentée, sauf dans MongoDB, où ce type de recherche est possible mais impraticable dans les faits à cause de sa lenteur.

Indexation sur schema-less

Les principes d'indexation se ressemblent, mais les moteurs NoSQL présentent néanmoins une différence importante avec les moteurs relationnels : leur schéma de données est libre. Dans un moteur relationnel, l'indexation est aisée parce que le schéma est fixé, et les colonnes sont visibles, en tant que métadonnées, par le moteur. Indexer une ou plusieurs colonnes revient, dans un moteur relationnel, à définir un index sur des colonnes bien connues. Dans un moteur NoSQL, personne ne sait si la valeur à indexer dans un index secondaire est bien présente dans, disons, tous les documents d'une collection. MongoDB, par exemple, permet la création d'index secondaires et ajoute des `NULL` dans l'index pour les documents qui ne contiennent pas la valeur à indexer, ou les ignore simplement si l'index est *sparse*.

Sans l'utilisation de ces index, les recherches complexes (celles qui incluent un élément qui ne constitue pas une clé ou plusieurs critères) se révèlent très lentes dans la plupart des systèmes. Les seuls moyens à disposition sont de travailler en mémoire ou de distribuer le travail, deux méthodes qui sont déjà utilisées dans les moteurs relationnels à travers le buffer et le partitionnement des données. À la rigueur, dans ce cas de figure, le partitionnement d'un moteur relationnel et le sharding d'une base NoSQL sont équivalents. La figure 4-3 présente les différences entre un moteur relationnel et un moteur NoSQL.

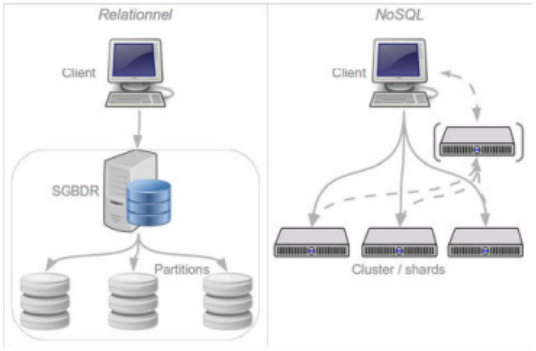


Figure 4-3
Partitionnement relationnel et NoSQL

Pour interroger un système relationnel, vous envoyez une requête au serveur qui déclenche un scan de la table. Le moteur de stockage parallélise les lectures sur les différents disques physiques qui contiennent des partitions de la table. Si les données sont déjà dans le buffer, aucune lecture sur disque n'est nécessaire.

Pour interroger un système NoSQL tel que MongoDB, le client fait d'abord appel au superviseur ou au répartiteur de charge (le démon `mongos` dans MongoDB).

Systèmes décentralisés

Tous les systèmes NoSQL n'imposent pas un contrôleur ou un point d'entrée unique. Riak, par exemple, est complètement décentralisé. C'est pour cette raison que nous avons mis cet élément entre parenthèses dans notre schéma.

Une fois les informations de sharding récupérées, le système lance une suite de requêtes aux différents serveurs qui contiennent les shards à rechercher. Cette fois, la répartition n'est pas décidée par le serveur, mais au niveau du client qui lance les requêtes et agrège les résultats. Ce mécanisme n'est pas plus rapide que celui des bases relationnelles. Dans un cas comme dans l'autre, malgré la répartition, le parcours sera coûteux et il pourra l'être parfois *à cause* de la répartition. Imaginez que vous lanciez une recherche sur un MongoDB shardé. La recherche s'effectue sur une dizaine de machines, mais l'une d'elle a un problème technique ou est surchargée. Les autres machines auront retourné leur résultat et le client attendra sur la dernière pour clore son travail. Nous avons à nouveau un goulet d'étranglement.

Les bases orientées documents

Les bases de données orientées documents présentent une structure différente par rapport aux bases en paires clé-valeur, principalement la structure de leurs lignes. Une base de données dans un entrepôt orienté documents peut être comparée à une table dans un SGBDR. La base contient des documents qui sont comme les lignes de cette table. Chaque document contient en général une clé unique qui l'identifie. Les deux bases de données orientées documents les plus populaires sont CouchDB et MongoDB.

Par « documents », il ne faut pas entendre documents binaires. Au contraire, ces documents sont structurés et lisibles par le moteur NoSQL. Le format privilégié pour le document est en général du JSON, un format de définition de données qui contient une structure hiérarchique de propriétés et de valeurs. Ces dernières peuvent être scalaires ou contenir des tableaux ou des sous-documents, et on peut souvent y attacher des fichiers binaires (voir section suivante).

Un autre avantage des bases orientées documents est le *versioning*. Dans un moteur relationnel, si vous devez changer la structure des tables pour faire évoluer votre application, vous devrez changer votre schéma et faire des choses horribles comme ajouter un numéro de version dans le nom même de la table. Par exemple, si vous avez créé une table `TarifIntervention` en 2011, et que votre direction estime en 2012 que la gestion des tarifs doit changer, vous allez probablement créer une table `TarifIntervention2`, parce que vous devez maintenir en parallèle l'ancienne et la nouvelle gestion des tarifs. La gestion d'une telle base de données deviendra de plus en plus complexe au fil du temps. Avec un moteur de bases de données orienté documents, vous pouvez simplement insérer des nouveaux documents comportant les modifications de structure, et maintenir dans le document un numéro de version, qui sera testé dans l'application cliente pour une prise en charge adaptée.

JSON

JSON *(JavaScript Object Notation)* est un format de représentation de données structurées. Il a été créé par Douglas Crockford, un développeur américain très impliqué dans le développement du langage JavaScript. C'est un format qui permet aussi bien de sérialiser l'état des classes que de structurer des données hiérarchiques ou non. Cette description devrait vous faire penser à XML. En effet, JSON remplit le même cahier des charges que le langage XML, au moins à son niveau basique. Il partage avec ce dernier son format pur texte, donc reconnaissable sur toute plate-forme et par tout système, la capacité d'exprimer des relations hiérarchiques et la facilité de traitement (génération et parsing) automatisé. En revanche – et c'est une des raisons pour lesquelles il est préféré à XML dans les bases NoSQL – JSON est beaucoup plus compact, moins verbeux et beaucoup plus simple à lire et à manipuler. XML est un langage de balisage qui dérive du langage SGML (*Standard Generalized Markup Language*, langage de balisage généralisé standard). Il permet de définir la structure, et éventuellement le formatage de documents potentiellement complexes, et de faciliter leur échange et leur manipulation. XML est très intéressant pour contenir et manipuler des documents, mais il peut être un peu trop lourd pour manipuler des données simples. La syntaxe et les structures de données de JSON, en revanche, proviennent d'un langage de programmation moderne, le JavaScript, ce qui en est la forme normée. Avoir dérivé une notation de données d'un langage de programmation orienté objet et dont la syntaxe, dérivée du C, est comprise par une majorité de programmeurs, assure une proximité de ce format avec des structures présentes dans les langages de programmation eux-mêmes, et donc une plus grande simplicité de parsing par rapport au XML. De plus, en étant moins chargé par l'expression des métadonnées, il est plus lisible à l'œil nu.

Le code suivant est un exemple de document JSON basé sur notre application web de test : un site d'information communautaire nommé Passerelles, qui souhaite publier des articles sur différents sujets, rédigés par les membres du site, et que les lecteurs peuvent évaluer et commenter. Ce document JSON stocke un article dont le titre est « Pourquoi les éléphants ont-ils de grandes oreilles ? » et qui comporte déjà un commentaire.

```
{
«auteur»:
{
    «prénom»:»Annie»,
    «nom»:»Brizard»,
    «e-mail»:»annie.brizard@cocomail.com»
},
«titre»:»pourquoi les éléphants ont-ils de grandes oreilles?»,
«mots-clés»:
[
    «faune»,
    «afrique»,
    «questions intriguantes»
],
«catégories»:
[
    «sciences»,
    «éducation»,
```

```
    «nature»
],
«date_création»: «12/05/2012 23:30»,
«statut»:»publié»,
«nombre_de_lectures»:»54»,
«commentaires»:
[
    {
        «date»:»14/05/2012 10:12»,
        «auteur»:»fred92@jmenmail.fr»,
        «contenu»:»il n'y a pas que les éléphants qui ont des grandes oreilles!»
    }
]
}
```

Par comparaison, le même document en XML donnerait quelque chose comme ceci :

```
<?xml version = «1.0» encoding = «utf-8»?>
<article>
<auteur prénom=»Annie» nom=»Brizard» e-mail=»annie.brizard@cocomail.com» />
<titre>pourquoi les éléphants ont-ils de grandes oreilles?</titre>
<mots-clés>
<mot-clé>faune</mot-clé>
    <mot-clé>afrique</mot-clé>
    <mot-clé>questions intriguantes</mot-clé>
</mots-clés>
<catégories>
    <catégorie>sciences</catégorie>
    <catégorie>éducation</catégorie>
    <catégorie>nature</catégorie>
</catégories>
<date_création>12/05/2012 23:30</date_création>
<statut>publié</statut>
<nombre_de_lectures>54</nombre_de_lectures>
<commentaires>
    <commentaire>
        <date>14/05/2012 10:12</date>
        <auteur>fred92@jmenmail.fr</auteur>
        <contenu>[CDATA[il n'y a pas que les éléphants qui ont des grandes
oreilles !]]</contenu>
    </commentaire>
</commentaires>
</article>
```

Formater du JSON

Les documents JSON retournés par les bases NoSQL ne sont bien entendu pas formatés comme dans les exemples de ce livre. Si vous souhaitez récupérer un document JSON et le formater manuellement pour en faire un copier-coller visuellement attractif, vous pouvez utiliser des formateurs JSON disponibles sur le Web. Citons par exemple JSON Format (http://jsonformat.com/) ou JSON Formatter (http://jsonformatter.curiousconcept.com/).

Consultez le site http://www.json.org/json-fr.html, pour visualiser la liste des types de données reconnus par JSON.

Les bases orientées colonnes

Les bases de données orientées colonnes (*Column Store* ou *Wide Column Store*) sont assez proches conceptuellement des tables relationnelles : elles comportent des colonnes avec un type de données. La structure des bases orientées colonnes est modélisée selon BigTable, la base de données de Google.

Une table comporte des clés, souvent appelées *rowkeys*, les clés de lignes. Ces clés sont uniques et sont maintenues dans un ordre lexicographique, c'est-à-dire un tri binaire des octets, sans interpréter la valeur qui s'y trouve, comme ce serait le cas dans un tri sur un entier en 32 bits, par exemple. Le tableau 4-1 présente des clés entières maintenues dans un ordre lexicographique.

Tableau 4-1. Clés dans un ordre lexicographique

Clé
1
10
22
3
45
9

À l'intérieur de la table, des familles de colonnes sont définies et regroupent des colonnes. La famille de colonnes est prédéfinie, et on lui attribue souvent des options, alors que les colonnes qui s'y trouvent ne sont pas prédéfinies, c'est-à-dire qu'il n'y a pas de description de schéma à l'intérieur d'une famille de colonnes. D'une ligne à l'autre, les colonnes présentes dans une famille peuvent varier selon les données à stocker.

Gestion des NULL

Dans une base relationnelle, la structure d'une table est fixée à l'avance, et toutes les colonnes sont référencées dans chaque ligne. Si une valeur de colonne est inconnue, le SGBDR inscrit un marqueur `NULL`. Dans une base orientée colonnes, la valeur non présente ou inconnue n'est simplement pas mentionnée dans la ligne. De cette façon, elle ne consomme aucun espace. Elle n'est simplement même pas présente. Le schéma varie donc de ligne en ligne.

Le stockage sur le disque est organisé par famille de colonnes. On peut donc considérer les familles de colonnes comme des sous-tables. Si on représente cette organisation du point de vue de la structure des données qu'on peut manipuler dans un langage comme Java, on peut voir les choses comme illustré à la figure 4-5.

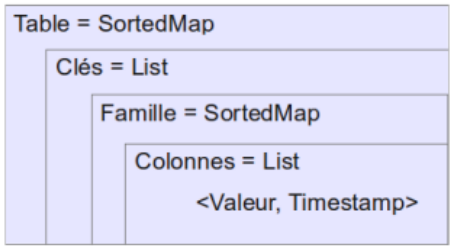


Figure 4-5
Structure du stockage orienté colonnes

La table est un ensemble trié de clés. La clé est une liste de familles (clé = nom de la famille, valeur = contenu), la famille un ensemble trié de colonnes, et la colonne une liste de valeurs-timestamp. Cela donne en une ligne :

```
SortedMap<Cle, List<SortedMap<Colonne, List<Valeur, Timestamp>>>>
```

Les colonnes comportent donc un nom de colonne, une valeur et un horodatage (timestamp), comme nous l'avons déjà vu.

Les bases orientées colonnes sont plus difficiles et complexes à mettre en place que les bases orientées documents ou paires clé-valeur, elles correspondent à des besoins plus larges, ce qu'on appelle le Big Data. Elles sont surtout déployées dans des compagnies qui manipulent de très importants volumes de données.

Leur stockage est autant que possible optimisé : les familles de colonnes sont stockées ensemble et compressées.

Les documents binaires

Les bases NoSQL peuvent être plus intéressantes que les bases relationnelles pour stocker des documents binaires. Traditionnellement, le stockage de documents binaires a été une problématique

vive des bases de données relationnelles. Nous avons vu que les moteurs relationnels sont optimisés pour le traitement et la restitution des données discrètes, de taille raisonnable et fortement typées. Pour assurer de bonnes performances et une bonne qualité de données, les principes de normalisation s'assurent que les données sont les plus atomiques possible. À partir de cette organisation, le moteur de stockage d'un SGBDR organise le stockage sur disque en allouant des pages de données de taille fixe, qui contiennent les lignes de table ou d'index. Cette organisation fonctionne bien pour des données structurées et très mal pour des objets larges. Les utilisateurs de SGBDR ont donc privilégié le système de fichiers pour stocker les documents, en insérant simplement le lien vers le fichier dans la base de données. Des mécanismes intermédiaires ont vu le jour, comme le type `DATALINK` de la norme SQL[7].

Comme les moteurs NoSQL ont une approche beaucoup plus souple et agnostique des données stockées, il n'y a théoriquement pas de raison d'éviter de stocker des documents binaires directement dans le stockage de la base. Les interfaces d'accès aux bases NoSQL sont aussi plus souples que le langage SQL pour ce genre d'approche. Pour transmettre des objets larges à l'aide de requêtes SQL, il faut réaliser des contorsions à l'aide d'extensions du langage. Les pilotes NoSQL sont en général beaucoup plus proches du langage client et permettent d'envoyer directement des objets. Plusieurs moteurs NoSQL comme CouchDB ou Riak offrent également des interfaces HTTP en REST qui permettent d'envoyer un document large. Comme ils sont conçus pour stocker de larges volumes de données de façon distribuée, beaucoup les utilisent pour stocker des binaires sans problème. D'autres moteurs, comme MongoDB ou Riak, offrent des couches spécialement destinées au stockage de binaires, comme GridFS pour MongoDB.

Pourquoi utiliser les bases de données NoSQL pour stocker des binaires plutôt que de passer pas le système de fichiers ? Simplement parce qu'il est beaucoup plus difficile d'implémenter un système de fichiers largement distribué et redondant, ce qu'offrent nativement les bases NoSQL. Autant profiter de ces caractéristiques pour les binaires également.

Le stockage du Big Data analytique

Depuis les premières implémentations des moteurs relationnels, les systèmes de gestion des données séparent clairement le stockage des données de leur traitement. La séparation entre moteur relationnel et moteur de stockage dans les SGBDR permet d'isoler deux responsabilités bien distinctes : l'interprétation et l'exécution de code de manipulation (recherche et modification) des données, et la gestion physique du stockage et de la cohérence. Pour aborder le Big Data analytique, cette séparation est également importante. Il faut clairement isoler les deux besoins pour bâtir une architecture solide.

Nous l'avons vu, le paradigme MapReduce est basé sur la programmation fonctionnelle. Si on dégage l'essence de ce qu'on fait avec Hadoop, c'est d'envoyer des fonctions aux données. Tout le framework Hadoop est simplement ça : envoyer des fonctions (du code compilé, qui contient deux fonctions ou méthodes : `map()` et `reduce()`). La puissance de ce modèle, c'est que vous avez pour vous toute la richesse du langage Java et ses innombrables bibliothèques à disposition. Dans vos

7. Voir par exemple : http://wiki.postgresql.org/wiki/DATALINK

fonctions, vous pouvez donc faire tout ce que vous voulez de vos données, et y accéder quel que soit leur format, de simples fichiers sur HDFS ou le contenu de bases de données comme Hbase.

Donc pour Hadoop, peu importe le stockage. Lorsque vous montez votre environnement Big Data, vous allez choisir le modèle ou le moteur de stockage qui répond le mieux à vos besoins. Rien ne vous empêche de stocker tout bêtement des fichiers CSV sur HDFS, ou le contenu brut des logs que vous voulez analyser par la suite. Dans les faits, il faut quand même un minimum de structure. Nous allons aborder le sujet en présentant quelques formats adaptés à la galaxie Hadoop.

Les contraintes de stockage pour Hadoop

Vous pouvez très bien travailler avec des fichiers bruts importés dans votre système. Attention, pour s'assurer de bonnes performances avec Hadoop, il ne faut pas travailler sur des petits fichiers. L'erreur à ne pas commettre est d'accumuler sur HDFS des fichiers incrémentaux d'importation de données de quelques Mo ou dizaines de Mo et de travailler à partir de cette source pour les tâches MapReduce. Cela pour deux raisons.

Premièrement, HDFS n'est pas du tout optimisé pour travailler avec des fichiers de petite taille, mais beaucoup plus apte à gérer des fichiers de grande taille. L'unité de stockage de HDFS est le bloc, dont la taille est par défaut de 64 Mo (129 Mo dans la configuration de HDFS avec Hadoop). HDFS ne peut pas déposer plusieurs fichiers dans un bloc, il peut en revanche découper un fichier de grande taille en plusieurs blocs. Si vous stockez une grande quantité de fichiers de petite taille (on parle de Big Data, et par conséquent de grands volumes), vous consommez donc un grand nombre de blocs.

Cela n'influe pas sur le stockage effectif sur le disque physique, car HDFS ne dimensionne pas ses fichiers réeels sur la taille du bloc. En revanche, le Namenode (le maître du cluster HDFS) conserve en mémoire un objet qui pointe sur chaque fichier et sur chaque bloc, dont la taille est d'environ 150 octets. Pour chaque fichier inférieur à un bloc, il lui faut maintenir en mémoire 300 octets pour le fichier et son bloc. Un million de fichiers, cela veut donc dire 3 Go en mémoire sur le Namenode. Pour plus de détails, consultez cette entrée de blog. http://blog.cloudera.com/blog/2009/02/the-small-files-problem/.

Deuxièmement, les tâches MapReduce ne travaillent pas avec plusieurs fichiers en même temps. Donc, charger une grande quantité de petits fichiers demande plus de tâches MapReduce, donc plus de nœuds, et plus d'attente sur des traitements séquentiels, ce qui se traduit par des diminutions importantes de performances. Il est donc important d'éviter les petits fichiers.

Le problème avec HDFS

Il y a une autre dimension à ce problème. À la base, un fichier écrit dans HDFS était immutable. HDFS ne supportait pas l'ajout de données à l'intérieur d'un fichier existant. Vous l'écrivez, vous le supprimez, mais vous ne pouvez pas le changer. N'oublions pas que HDFS est conçu en premier lieu pour stocker des données analytiques : on écrit une fois, on distribue le fichier sur le cluster, et on peut ensuite lire à l'infini, en parcourant les fichiers plutôt que par des accès aléatoires comme ce serait le cas si on faisait une recherche dans le contenu.

Les versions récentes de HDFS implémentent la capacité d'ajouter des données à la fin des fichiers existants (donc faire un *append*). Mais il a fallu résoudre des problèmes techniques. Devinez. Les mêmes qui se posent dans les moteurs NoSQL distribués, c'est-à-dire comment assurer la cohérence des fichiers répliqués, comment revenir en arrière si une écriture se passe mal, etc. Il a donc fallu ajouter des états supplémentaires dans l'algorithmique de l'écriture et mettre en place un système de verrouillage. La fonctionnalité a été distribuée en 2008 dans la version 1.19.0 de HDFS, puis retirée parce qu'elle posait un certain nombre de problèmes, puis réactivée après correction dans la version 0.21.0 (pour plus de détails sur ce problème, vous pouvez vous référer à la demande HDFS-256 sur le Jira de la fondation Apache : https://issues.apache.org/jira/browse/HDFS-265).

L'append étant maintenant implémenté, il reste toujours une limitation : on ne peut toujours pas réaliser des modifications aléatoires à l'intérieur d'un fichier. On peut donc écrire des fichiers de type log (comme des structures LSM – *Log-Structured Merge-Tree* – ou le journal de type *write-ahead log* de Hbase), mais pas maintenir des fichiers de données basés sur une allocation différemment structurée.

Pour ces différentes raisons, il y a une petite histoire de réflexion et de solution autour du meilleur moyen de stocker ses données pour les traiter avec Hadoop, Hive ou maintenant des frameworks comme Spark. Une première solution est bien sûr de générer vous-même des fichiers plus importants pour les stocker dans HDFS. En général, vous manipulez des données structurées. Le CSV est une solution limitée techniquement, le XML est trop verbeux, le JSON peut faire l'affaire, mais on peut espérer de meilleures performances. Vous pouvez donc structurer vos fichiers avec Thrift, Protobuf ou Avro.

Mais attendez, vous avez en fait l'embarras du choix. Plusieurs formats ont été conçus pour adresser ce besoin, il y a eu plusieurs initiatives qui ont développé des solutions maintenant largement utilisées.

Le SequenceFile

En interne, Hadoop MapReduce doit échanger des informations structurées. Entre la phase Map et la phase Reduce, les données doivent être triées et envoyées aux reducers. Cette phase s'appelle *Shuffle and Sort* (déplacer et trier). Le format utilisé en interne par Hadoop pour ce faire s'appelle un SequenceFile. C'est simplement un fichier binaire contenant des paires clé-valeur.

MapReduce fournit une bibliothèque qui permet très simplement de travailler avec des SequenceFiles, ce qui en fait un format très utilisé en entrée-sortie pour des opérations MapReduce. Le format est simple et décrit dans la documentation d'Hadoop (http://wiki.apache.org/hadoop/SequenceFile). Il peut être compressé ou non. La compression est assurée par un algorithme (nommé un codec) qui implémente l'interface `org.apache.hadoop.io.compress.CompressionCodec`. Les classes existantes implémentent la compression en Zlib, BZip2, Gzip, Snappy : classes `DefaultCodec`, `BZip2Codec`, `GzipCodec`, `SnappyCodec` (https://code.google.com/p/hadoop-snappy/). Comme son nom l'indique, ZLib est le codec par défaut. Notons qu'un des avantages de `BZip2Codec` est qu'il permet de découper le travail de compression/décompression (il est splittable), et qu'il peut donc être mis en œuvre en parallèle par plusieurs tâches MapReduce. Cela a son avantage sur des fichiers de grande taille. On va y revenir.

Le fichier a bien sûr un en-tête, qui indique notamment le codec de compression qui a été utilisé, puis un contenu, composé de paires clé-valeur, la valeur binaire étant compressée le cas échéant.

La classe `org.apache.hadoop.io.SequenceFile` (http://hadoop.apache.org/docs/stable/api/src-html/org/apache/hadoop/io/SequenceFile.html) fournit trois sous-classes qui prennent en charge la manipulation des fichiers : `SequenceFile.Writer`, `SequenceFile.Reader` et `SequenceFile.Sorter`. Les noms sont autodescriptifs.

Voici un exemple très simple de manipulation d'un SequenceFile à l'aide de l'API Hadoop, en Java :

```
Configuration conf = new Configuration();
Path path = new Path("/chemin_vers/sequencefile");
SequenceFile.Reader reader = new Reader(conf, Reader.file(path));
WritableComparable key = (WritableComparable) reader.getKeyClass().newInstance();
Writable value = (Writable) reader.getValueClass().newInstance();

while (reader.next(key, value)) {
    // faites quelque chose ici;
}
reader.close();
```

Cet exemple correspond à la manière classique d'itérer dans une collection de données provenant d'un moteur de bases de données, à l'aide d'une forme de curseur.

Pour résoudre le problème des petits fichiers que nous avons évoqué plus haut, l'une des solutions les plus répandues est d'utiliser des SequenceFiles, qui sont conçus pour ça : les fichiers sont encapsulés dans un fichier SequenceFile plus grand, le nom du fichier et d'autres métadonnées au besoin sont inclus dans la clé, et le contenu du fichier est stocké et compressé dans la valeur de chaque paire clé-valeur. L'avantage du SequenceFile est qu'il est découpable par l'implémentation de MapReduce : Hadoop est donc capable de distribuer des parties du SequenceFile sur plusieurs tâches MapReduce.

Le RCFile

RCFile *(Record Columnar File)* est une structure qui prend en charge le stockage distribué de données tabulaires. Il a été développé conjointement par Facebook, l'université de l'État de l'Ohio et l'Institute of Computing Technology de l'Académie des Sciences chinoise. En 2011, un papier a été publié et présenté à l'ICDE '11 *(International Conference on Data Engineering)* : «*RCFile: a Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse systems*» Vous le trouvez facilement. Son objectif est de remplacer les formats de stockage traditionnels (tabulaire, orienté colonne, etc.) afin d'être parfaitement adapté au traitement sur Hadoop. Lorsque l'article est paru, RCFile était déjà utilisé pour le décisionnel de Facebook, ainsi qu'adopté dans Hive et Pig, deux sur-couches analytiques pour Hadoop.

Les principes de base sont les suivants : les données sont représentées de façon tabulaire, avec des colonnes et des lignes. Chaque ligne est conservée sur le même nœud. Le partitionnement des données est donc effectué par ligne. En revanche, RCFile compresse ses données pour des performances accrues, et il le fait par colonne, ce qui offre un meilleur taux de compression

(puisqu'une compression par dictionnaire appliquée sur les données d'une même colonne trouvera plus de valeurs semblables à résumer que si on appliquait une compression par ligne).

RCFile stocke ses tables dans des blocs HDFS, en composant des groupes de lignes *(row groups)* dans chaque bloc. Les groupes de lignes sont toutes d'une taille identique sur la même table, et un bloc HDFS peut en contenir plusieurs. Chaque groupe organise son stockage en colonnes, c'est-à-dire que les cellules sont regroupées et compressées par colonnes. Outre l'avantage en termes de compression, cela permet lors de la lecture pour le job MapReduce de n'accéder qu'aux colonnes demandées par la requête. Le stockage par colonne est particulièrement bien adapté à l'analytique, qui va traiter toutes les lignes de la table, mais seulement les colonnes nécessaires à l'analyse en cours. De plus, les valeurs ne sont décompressées que si la requête en a vraiment besoin (une fonctionnalité baptisée *lazy decompression*).

Requêtes

Nous parlons ici de requête. RCFile est particulièrement adapté aux outils de requêtages par-dessus Hadoop comme Hive ou Pig. En Hive, la demande est exprimée en langage SQL (HiveQL). Une requête telle que `SELECT produit, quantite FROM ventes WHERE annee = 2008`, entraînera la récupération seulement des colonnes `produit`, `quantite` et `annee`, et la décompression des données des colonnes `produit` et `quantite` qui correspondent à l'année 2008.

Par nécessité – nous venons de voir les limitations de HDFS –, l'écriture n'est disponible qu'en mode append, et la structure interne du RCFile doit se débrouiller avec ça.

L'accès aux données à l'intérieur des fonctions MapReduce se fait à l'aide des classes `RCFileInputFormat`, `RCFileRecordReader` et `RCFileOutputFormat`. Pour un exemple de code utilisant Hive, vous pouvez vous référer à cette entrée de blog : http://sumit1001.blogspot.fr/2012/01/rcfile-fast-and-space-efficient-data.html.

L'ORC File

Nous n'avons pas encore tout à fait fini. Restons sur notre ligne chronologique. RCFile a apporté des optimisations bienvenues au stockage pour Hadoop, mais il restait quelques imperfections. D'abord, bien que RCFile adopte un modèle tabulaire, les colonnes ne sont pas typées. Elles contiennent donc des données purement binaires qu'il faut convertir dans son application en entrée comme en sortie. Cela peut avoir quelques désavantages, par exemple si on souhaite implémenter une indexation sur les valeurs, ou profiter de types de compression adaptés à des types de données spécifiques. L'implémentation elle-même de RCFile pouvait encore être améliorée, notamment en ce qui concerne la recherche dans le fichier et la compression.

Pour ces différentes raisons, un projet de la fondation Apache a créé une version améliorée de RCFile, nommée ORC File (pour *Optimized Row Columnar File*), spécifiquement pour l'intégration avec Hive. L'objectif était d'améliorer les performances et le stockage, tout en prenant directement en charge les types de données manipulés par Hive, notamment les types complexes comme les listes ou les maps, et les types de bases de données que Hive reconnaît en implémentant la norme SQL (Hive essaie par design de s'approcher de la norme SQL-92).

Le format de fichier inclut des innovations, comme une séparation des groupes de lignes en morceaux plus volumineux nommés stripes (256 Mo par défaut), l'ajout d'une structure de fin (un *footer*, en plus donc de l'en-tête) au niveau des stripes et du fichier entier avec quelques agrégats calculés sur les colonnes, ainsi qu'une indexation interne qui accélère l'accès aux groupes de lignes, en permettant d'éliminer des lignes à parcourir par blocs de 10 000 lignes par défaut.

Indexation

En maintenant les valeurs minimale et maximale de chaque bloc de 10 000 lignes, ces blocs peuvent être éliminés pour résoudre une requête qui cherche hors de la plage de valeurs de ce bloc.

La compression est adaptée à chaque type de données stocké dans les colonnes et elle est optimisée. Chaque colonne présente plusieurs flux *(streams)* de données qui contiennent non seulement la valeur de la cellule, bien sûr, mais aussi des structures pour accélérer les lectures. Par exemple, les colonnes de type string sont implémentées sous forme d'un dictionnaire, ce qui unifie chaque valeur et compresse d'autant plus la colonne et accélère les recherches sur les valeurs. Elles ont quatre streams: un stream de données (les valeurs de la colonne), un stream de bits de présence (pour déterminer rapidement si la colonne est en `NULL` ou non), un stream des données du dictionnaire et un stream contenant la longueur de chaque entrée du dictionnaire.

ORC File est spécifiquement dédié à Hive, et c'est un ajout récent à Hive qui fait partie de l'initiative destinée à améliorer ses performances. Les gains de performances constatés sur Hive avec ORC File par rapport aux autres modes de stockage sont impressionnants. Vous pouvez faire appel à ORC File sans passer par Hive en utilisant Hcatalog.

Hcatalog

Sur des systèmes analytiques, vous accédez aux données avec un modèle à deux ou trois couches: il vous faut un outil de requêtage, que ce soit MapReduce, Hive ou toute autre interface (elles fleurissent), et des données stockées. Vous pouvez aussi utiliser une couche intermédiaire, une couche d'abstration pour le format ou la structure des données. On en connaît beaucoup. Dans le monde relationnel, on pense aux univers de Business Object.

En Hadoop, on a déjà Hive, qui remplit ce rôle en offrant une représentation en tables relationnelles de données stockées en HDFS. Hcatalog est une de ces couches d'abstraction, qui représente sous forme tabulaire et relationnelle des données sous-jacentes de fichiers plats sur HDFS, de SequenceFile, de RCFile ou d'ORC Files. Pig et MapReduce peuvent utiliser Hcatalog comme source de données. En l'utilisant, vous pouvez donc interfacer Pig ou mapReduce avec ORC File.

Le format a été récemment travaillé pour améliorer encore les performances, en ajoutant une vectorisation des valeurs pour pouvoir les analyser en mode batch à l'aide de Hive. En un mot, Hive est maintenant capable de récupérer des lignes par groupes de 1024 et de stocker les valeurs en mémoire dans des vecteurs (des tableaux de valeurs simples) pour pouvoir les traiter beaucoup plus rapidement avec des algorithmes simplifiés. Pour plus d'informations, vous pouvez consulter la documentation de Hive à cette adresse: https://cwiki.apache.org/confluence/display/Hive/Vectorized+Query+Execution.

Parquet

Nous n'en avons pas tout à fait fini avec la quête pour le bon format de stockage sur Hadoop, ni avec les formats de stockage tabulaire. En 2012, Twitter et Cloudera décident de travailler ensemble pour développer un format de stockage tabulaire basé sur HDFS et destiné à optimiser le stockage et les performances. Impression de déjà-vu ? En juillet 2013, la version 1 de leur développement sort, baptisée Parquet. En 2014, Parquet rejoint la longue liste des projets en incubation Apache (http://parquet.incubator.apache.org/), et la version 2 de Parquet est depuis en développement sous le giron de la fondation.

Parquet est premièrement un format de fichier qui permet de stocker des structures imbriquées complexes en implémentant l'algorithme décrit par Google dans son papier sur Dremel. Nous avons déjà parlé de cet algorithme. Parquet est donc l'implémentation libre de cet aspect de Dremel et il est particulièrement ciblé pour l'interrogation interactive des données Big Data, principalement pour Cloudera en utilisant leur outil de requêtage nommé Impala (http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html).

Comme dans ORC File, la compression est choisie individuellement par colonne. Le stockage interne n'est pas tellement éloigné de RCFile, avec une stratégie de groupes de lignes *(row groups)* et de stockage par colonne *(column chunks)* pour optimiser la compression et l'accès. Le pied de fichier *(footer)* contient les informations de métadonnées.

Le stockage est défini par le projet parquet-format (https://github.com/Parquet/parquet-format). Les métadonnées du format de stockage sont encodées en thrift. En utilisant le fichier thrift de définition, vous pouvez générer du code de gestion pour lire et écrire en Parquet. Dans les faits, ce n'est pas nécessaire. Le projet parquet-mr fournit le nécessaire pour utiliser directement des fichiers Parquet en MapReduce, tandis que le concept de modèles d'objet (*object models*) et ses implémentations de convertisseurs offrent déjà les passerelles nécessaires pour représenter les données Parquet en Thrift, Avro, Protobuf, Hive, Pig, etc.

Parquet est un projet déjà mature, très intéressant et prometteur. C'est un excellent moteur de stockage à considérer si vous démarrez un projet Big Data analytique.

Partie II

Panorama des principales bases de données NoSQL

En regard de ce que nous avons appris dans le chapitre précédent, nous allons présenter dans cette partie les différents types de bases de données NoSQL ainsi que les produits qui ont été développés autour d'eux. Nous ne parlerons que des implémentations libres, car elles constituent de toute façon l'essentiel des outils NoSQL utilisés par les grandes sociétés. Chaque grand moteur NoSQL (MongoDB, Cassandra, HBase, etc.) fera l'objet d'un chapitre.

Nous étudierons ces SGBD NoSQL de façon pratique : nous verrons brièvement comment les installer, quels en sont les outils et les fonctionnalités d'administration, et comment les manipuler. Pour cela, nous utiliserons un système Linux, en l'occurrence Ubuntu Server, dans sa version 14.04 LTS (surnommée *The Trusty Tahr*). LTS signifiant *Long Term Support* (maintenue à long terme), cette version sortie en avril 2014 sera supportée par Canonical jusqu'en avril 2019, ce qui assure une relative pérennité à nos propos.

Préliminaires

Pour montrer comment fonctionnent les différents moteurs NoSQL, nous utiliserons un langage client lisible et facile à comprendre par tout développeur. Il s'agit de Python, un langage de script très populaire. Il comporte maintenant des bibliothèques d'interfaçage pour tous les serveurs NoSQL importants. Python 3.4 est installé par défaut dans Ubuntu 14.04. Depuis quelques versions, Ubuntu n'installe que Python 3 par défaut. La version 3 de Python apporte des changements très importants qui ne sont pas encore supportés par toutes les bibliothèques. Nous installons donc Python 2.7 pour une meilleure compatibilité, le paquet s'appelle simplement python. Par ailleurs, il nous faut installer `pip`, un programme de gestion de paquets Python :

```
sudo apt-get install python
sudo apt-get install python-pip
```

Nous allons utiliser la technique d'environnement virtuel de Python pour ne pas impacter

l'installation de base. Les paquets d'extension de Python que nous manipulerons peuvent s'installer soit dans le répertoire système (par exemple, `/usr/local/lib/python2.7/site-packages` sur Ubuntu), soit dans un répertoire propre à l'utilisateur, en créant une installation virtuelle (`virtualenv`) de Python, dans son répertoire `$HOME`. Toutes les bibliothèques système et les exécutables y sont représentés par des liens symboliques, et les additions de paquets y seront déposées, ils sont donc propres à l'utilisateur et ne troublent pas l'installation système. De plus, il n'y a pas besoin d'être utilisateur `root` pour les ajouter. À l'aide de `pip`, installons `virtualenv` :

```
sudo pip install virtualenv
```

Ensuite, nous créons l'environnement virtuel pour nos tests :

```
cd ~
mkdir python_env
cd python_env
virtualenv nosql
```

Nous voilà prêts à découvrir les différents moteurs NoSQL !

5

Hadoop et HBase

Hadoop

Hadoop est un projet socle, dont on peut dire qu'il est à l'origine d'une petite révolution informatique, et qui ne concerne pas directement les bases de données. La naissance d'Hadoop préfigure une liste de projets libres nés de publications de Google, notamment *MapReduce: Simplified Data Processing on Large Clusters* (MapReduce : traitement de données simplifié sur des clusters de grande taille, voir http://research.google.com/archive/mapreduce-osdi04-slides/index.html).

Hadoop est un projet en constante évolution. Comme nous l'avons déjà dit, il a subi un changement important dans sa version 2, puisque l'environnement de distribution des tâches a été séparé de l'implémentation de l'algorithme MapReduce. Au moment où nous écrivons ces lignes, la version stable est la 2.6.0, sortie en novembre 2014. Vous pouvez toujours télécharger la version précédente avec MapReduce intégré dans la branche des versions 1.2.

Installation

Cloudera, une société qui se spécialise dans le service autour d'Hadoop, offre une distribution de niveau entreprise de tout l'environnement Hadoop, avec une version libre (CDH) et une version entreprise commerciale. Nous allons utiliser la distribution libre ; au moment où nous écrivons ce chapitre, elle est en version 5.3.1 et intègre un certain nombre d'outils de la nouvelle génération Hadoop. L'installation de production sur plusieurs nœuds peut être effectuée avec un gestionnaire graphique que nous vous recommandons. Nous allons vous indiquer la marche à suivre pour réaliser cette installation. Nous vous présenterons ensuite une installation différente, afin d'effectuer nos tests en mode pseudo-distribué, c'est-à-dire en simulant un système complet sur une seule machine.

Installation de production

Note préliminaire

Installer Hadoop, HDFS et HBase n'est parfois pas de tout repos. Comme l'installation d'environnements Java complexes, vous pourrez être amené à rencontrer certains problèmes lors du processus d'installation. Même avec la distribution Cloudera, nous avons dû résoudre plusieurs problèmes lors de notre installation en mode pseudo-distribué sur une seule machine Ubuntu. Nous vous indiquerons quelques pistes de solution en encadré.

Nous nous rendons à l'adresse http://www.cloudera.com/content/cloudera/en/products-and-services/cdh.html. Vous pouvez tester une version Live de CDH sans installation à cette adresse : http://go.cloudera.com/cloudera-live.html. Un bouton permet d'installer automatiquement la version 4 de CDH grâce à l'outil Cloudera Manager (voir figure 5-1).

Figure 5-1
Téléchargement de Cloudera Manager

Nous cliquons sur le bouton pour lancer la procédure d'installation en téléchargeant Cloudera Manager.

64 bits obligatoire

Attention, Cloudera Manager ne fonctionne que sur un environnement 64 bits. Il vérifie aussi que vous utilisez bien une distribution supportée. Pour notre exemple, nous l'installons sur Ubuntu Server 14.04 LTS.

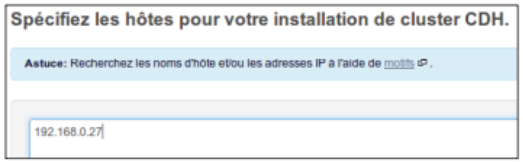
Nous téléchargeons et exécutons le gestionnaire sur notre serveur:

```
wget http://archive.cloudera.com/cm4/installer/latest/cloudera-manager-installer.bin
chmod u+x cloudera-manager-installer.bin
sudo ./cloudera-manager-installer.bin
```

Le gestionnaire d'installation démarre, télécharge les paquets nécessaires et les installe. Lorsqu'il a terminé, nous pouvons continuer l'installation à partir du manager lui-même, qui est une très belle application web, démarrée sur le port 7180. Donc, par exemple pour un accès en local, http://localhost:7180. L'utilisateur et le mot de passe par défaut sont `admin`. Une mise à niveau vers la version complète est proposée, nous restons sur la version gratuite, qui contient vraiment tout ce dont nous avons besoin à notre niveau.

L'étape suivante nous demande de spécifier l'adresse des nœuds. Il convient ici d'indiquer des noms de machines ou des adresses IP, séparés par des virgules, ou d'utiliser des motifs d'adresses IP. Pour notre exemple, nous indiquons l'adresse de notre seule machine. Attention à bien mentionner une IP externe et non la boucle locale (127.0.0.1), afin que HDFS ne s'emmêle pas les pinceaux par la suite. Il est important que le réseau et le nom de la machine soient bien configurés sur les nœuds dès le départ, pour que les noms de machines et les IP publiques correspondent. Nous indiquons donc l'IP publique de notre machine, comme illustré sur la figure 5-2.

Figure 5-2
Ajout des noms d'hôtes



Nous cliquons ensuite sur Recherche pour permettre la validation du ou des nœuds. Nous sélectionnons ensuite la version de CDH que nous souhaitons installer sur nos hôtes, soit CDH4, la dernière version en date. Nous validons en cliquant sur Continuer.

SSH

Dans la mesure où Cloudera Manager effectue ces opérations via SSH, assurez-vous que le démon SSHD tourne sur vos serveurs. Si vous utilisez une distribution Debian ou Ubuntu, vous pouvez l'installer grâce à la commande :

```
sudo apt-get install openssh-server
```

Nous fournissons ensuite l'identifiant de connexion SSH pour tous les nœuds. Il est tout à fait possible d'indiquer `root` ou un compte qui a les privilèges `sudo` sans mot de passe.

Être sudo sans saisir son mot de passe

Pour accepter qu'un utilisateur puisse lancer des commandes `sudo` sans mot de passe, vous devez l'indiquer dans le fichier `/etc/sudoers`. Pour modifier ce fichier avec l'éditeur par défaut, lancez la commande:

```
sudo visudo
```

Ajoutez à la fin du fichier une ligne avec le nom d'utilisateur que vous voulez configurer. Pour notre exemple, il s'agit de l'utilisateur `rudi`:

```
rudi ALL=(ALL) NOPASSWD: ALL
```

Pour l'identification du compte, nous avons eu des problèmes par une approche par mot de passe, nous avons donc opté pour une authentification SSH par clé. Pour ce faire, il faut créer une paire de clés sur la machine cliente (votre ordinateur de bureau) avec la commande suivante :

```
ssh-keygen -t dsa
```

qui va créer la clé privée et la clé publique dans des fichiers dont vous pouvez indiquer le nom. Par défaut, il s'agit des fichiers `/home/$USER/.ssh/id_dsa` et `/home/$USER/.ssh/id_dsa.pub`. Vous pouvez aussi protéger la clé privée par un mot de passe *(passphrase)* que vous devrez communiquer à Cloudera Manager pour qu'il ouvre la clé privée. La clé publique doit être placée sur le serveur, dans le fichier `/home/$USER/.ssh/authorized_keys`, `$USER` étant ici le compte du serveur que nous allons utiliser. La clé privée sera utilisée par Cloudera Manager pour signer la demande, et la signature sera authentifiée par le serveur en utilisant la clé publique. Il s'agit, comme vous le savez sans doute, d'une méthode traditionnelle d'authentification des clients SSH.

Les clés étant générées, nous ajoutons la clé publique sur le serveur en utilisant un script conçu pour cela :

```
ssh-copy-id -i ~/.ssh/id_rsa.pub "rudi@192.168.0.27"
```

Nous avons ici conservé le nom de la clé par défaut : `id_rsa`. Nous chargeons ensuite le fichier `.ssh/id_rsa` dans l'interface de Cloudera Manager, comme illustré sur la figure 5-3.

Figure 5-3
Saisie des identifiants de connexion SSH

Authentification par mot de passe

Si vous préférez essayer l'authentification par mot de passe, assurez-vous au moins que le serveur SSH accepte l'authentification interactive par clavier *(keyboard-interactive)*. Si ce n'est pas le cas, vous obtiendrez une erreur dans le Cloudera Manager (« keyboard-interactive auth not allowed by server »). Les messages d'erreur sont visibles dans le fichier de log situé à l'adresse `/var/log/cloudera-scm-server/cloudera-scm-server.log`.

Pour activer l'option d'authentification dans SSHD, modifiez le fichier `/etc/ssh/sshd_config` comme suit :

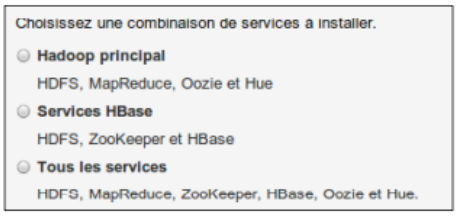
```
ChallengeResponseAuthentication=yes
```

puis redémarrez le démon SSHD :

```
sudo service ssh restart
```

À la fin de l'installation, tous les paquets de l'environnement sont installés, notamment Hue, une interface web de gestion d'Hadoop, et Flume-ng, un outil de traitement de log en MapReduce. Cloudera Manager affiche alors un résumé, puis demande de sélectionner les services à installer et démarrer (figure 5-4).

Figure 5-4
Choix de l'installation



Nous choisissons «Tous les services: HDFS, MapReduce, ZooKeeper, HBase, Oozie et Hue». Il convient ensuite de configurer les outils, principalement les répertoires utilisés. Pour notre démonstration, nous conservons les valeurs par défaut et nous cliquons sur Continuer. L'installation effectue la configuration (création des répertoires nécessaires aux services dans HDFS) et le démarrage des services. Si la configuration est correcte, aucun problème n'est rencontré.

Configuration du réseau

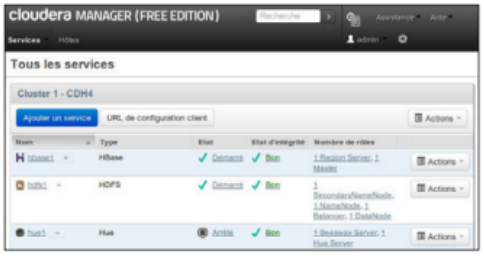
Au risque de nous répéter, soyez attentif à la configuration de votre réseau, et notamment à l'adresse de vos machines. Si le nom de vos nœuds est lié à la boucle locale dans le fichier `hosts`:

```
127.0.0.1 nom_du_noeud
```

le démarrage du service HDFS sera compromis.

Il est maintenant possible de gérer le serveur et les services à travers Cloudera Manager (figure 5-5).

Figure 5-5
Cloudera Manager



Installation manuelle

Vous pouvez bien sûr installer Hadoop manuellement pour obtenir la dernière version ou si votre distribution n'est pas supportée par ClouderaManager. La marche à suivre est indiquée en annexe.

Installation en mode pseudo-distribué

L'installation précédente permet de mettre en place un système avec plusieurs nœuds. Cloudera a également prévu une installation en mode pseudo-distribué. Les instructions complètes sont disponibles à l'adresse suivante : https://ccp.cloudera.com/display/CDH4DOC/Installing+CDH4+on+a+Single+Linux+Node+in+Pseudo-distributed+Mode.

Reprenons notre exemple depuis le début.

YARN

À l'heure actuelle, deux distributions sont possibles : une traditionnelle et une autre utilisant une nouvelle architecture de la couche MapReduce nommée YARN (MRv2). Nous allons utiliser la version traditionnelle, la MRv1, pour MapReduce version 1.

Java

Avant tout, nous devons installer Java, le JDK 1.6 dont Hadoop a besoin. Plusieurs méthodes sont possibles pour Ubuntu comme indiqué sur la page http://doc.ubuntu-fr.org/java. Nous effectuons ici une installation manuelle. Nous téléchargeons donc l'installeur depuis le site d'Oracle, puis nous l'installons :

```
sudo mkdir /usr/local/java
sudo cp jdk-6u35-linux-x64.bin /usr/local/java
cd /usr/local/java
sudo sh ./jdk-6u35-linux-x64.bin
```

La décompression du JDK dans le répertoire démarre. Le JRE est décompressé dans le répertoire `/usr/local/java/jdk1.6.0_35`. Nous allons ensuite ajouter les variables d'environnement nécessaires à l'exécution de Java. Nous pourrions les insérer dans le fichier `/etc/profile`, mais sur Ubuntu, `/etc/profile` source des fichiers individuels contenus dans `/etc/profile.d/`. Nous allons donc utiliser cette méthode en créant un fichier `jdk1.6.0_35.sh` dans ce répertoire :

```
sudo vim /etc/profile.d/jdk1.6.0_35.sh
```

Ici, nous utilisons l'éditeur Vim. S'il ne vous est pas familier, utilisez l'éditeur Nano qui est plus simple. Si vous travaillez en environnement graphique, lancez Gedit ou tout autre éditeur de votre choix.

Nous écrivons les lignes suivantes dans le fichier pour ajouter le répertoire du JRE dans le chemin et créer la variable d'environnement `JAVA_HOME` nécessaire à l'exécution des applications Java.

```
JAVA_HOME=/usr/local/java/jdk1.6.0_35
PATH=$PATH:$HOME/bin:$JAVA_HOME/bin
export JAVA_HOME
```

```
export PATH
```

Nous indiquons ensuite à Ubuntu quelle version de Java est installée et comment la trouver. Il s'agit ici d'une spécificité Ubuntu qui n'est bien sûr pas requise sur une autre distribution de Linux. La deuxième ligne indique à Ubuntu que notre installation est l'exécutable Java par défaut.

```
sudo update-alternatives --install «/usr/bin/java» «java» «/usr/local/java/
jdk1.6.0_35/bin/java» 1
sudo update-alternatives --set java /usr/local/java/jdk1.6.0_35/bin/java
```

Ces commandes créent un lien symbolique `/usr/bin/java` qui pointe sur `/etc/alternatives/java`, lui-même un lien symbolique sur notre binaire dans `/usr/local/java/jdk1.6.0_35/bin/java`.

Nous exécutons ensuite la commande suivante pour recharger le fichier `profile` sans avoir besoin de redémarrer Ubuntu :

```
. /etc/profile
```

Puis nous vérifions que la variable d'environnement `JAVA_HOME` est bien présente :

```
printenv | grep JAVA
```

Ce qui doit retourner cette ligne :

```
JAVA_HOME=/usr/local/java/jdk1.6.0_35
```

Nous testons aussi en appelant l'exécutable `java` :

```
java -version
```

Nous obtenons cette réponse :

```
java version «1.6.0_35»
Java(TM) SE Runtime Environment (build 1.6.0_35-b10)
Java HotSpot(TM) 64-Bit Server VM (build 20.10-b01, mixed mode)
```

Cloudera

Ensuite, nous téléchargeons un paquet qui configurera les dépôts pour la distribution de Cloudera. Ce paquet dépend bien sûr de votre distribution (Ubuntu Server 14.04 Trusty pour notre exemple). Les liens sont disponibles sur le site de Cloudera à l'adresse https://ccp.cloudera.com/display/SUPPORT/Downloads. Nous déposons tout dans un répertoire `cloudera` pour faire les choses proprement.

```
mkdir cloudera
cd cloudera
wget http://archive.cloudera.com/cdh4/one-click-install/precise/amd64/cdh4-
repository_1.0_all.deb
sudo dpkg -i cdh4-repository_1.0_all.deb
```

Le paquet ajoute les dépôts et la clé PGP d'authentification de la source. Nous installons ensuite le paquet Hadoop en configuration pseudo-distribuée :

```
sudo apt-get update
sudo apt-get install hadoop-0.20-conf-pseudo
```

qui récupère et installe toutes les dépendances. Une fois cela effectué et avant de démarrer Hadoop, nous devons formater le `namenode` (le service qui assure la coordination des fichiers) de HDFS :

```
sudo -u hdfs hdfs namenode -format
```

Erreur JAVA_HOME

Ici encore, il est possible que vous receviez une erreur de type « JAVA_HOME is not set ». Si c'est le cas, cela signifie que le `JAVA_HOME` défini dans les fichiers d'environnement de Hadoop ne pointe pas au bon endroit. Un bon moyen pour régler ce problème pour tout ce qui tourne autour d'Hadoop consiste à mettre la bonne valeur de `JAVA_HOME` dans le fichier `/etc/default/bigtop-utils` (voir http://bigtop.apache.org/). Pour notre exemple, nous ajoutons :

```
export JAVA_HOME=»/usr/local/java/jdk1.6.0_35»
```

```
à la fin du fichier.
```

Nous lançons les services HDFS pour éviter le redémarrage :

```
for service in /etc/init.d/hadoop-hdfs-*
> do
> sudo $service start
> done
```

Nous obtenons le résultat suivant :

```
 * Starting Hadoop datanode:
starting datanode, logging to /var/log/hadoop-hdfs/hadoop-hdfs-datanode-ubuntu-server-
64-fr.out
 * Starting Hadoop namenode:
starting namenode, logging to /var/log/hadoop-hdfs/hadoop-hdfs-namenode-ubuntu-server-
64-fr.out
 * Starting Hadoop secondarynamenode:
starting secondarynamenode, logging to /var/log/hadoop-hdfs/hadoop-hdfs-
secondarynamenode-ubuntu-server-64-fr.out
```

Nous vérifions que HDFS est lancé à l'aide de l'utilitaire `jps` :

```
sudo /usr/local/java/jdk1.6.0_35/bin/jps
```

Nous créons ensuite le répertoire `/tmp` dans HDFS :

```
sudo -u hdfs hadoop fs -mkdir /tmp
sudo -u hdfs hadoop fs -chmod -R 1777 /tmp
```

puis nous vérifions :

```
sudo -u hdfs hadoop fs -ls -R /
```

Nous obtenons alors :

```
drwxrwxrwt   - hdfs supergroup          0 2012-11-20 18:23 /tmp
```

ce qui prouve que HDFS fonctionne.

Nous devons encore créer les répertoires pour MapReduce dans HDFS et démarrer les démons MapReduce. Il s'agit de faire des copier-coller à partir de la documentation Cloudera. Nous listons simplement les commandes sans commentaire :

```
sudo -u hdfs hadoop fs -mkdir /var
sudo -u hdfs hadoop fs -mkdir /var/lib
sudo -u hdfs hadoop fs -mkdir /var/lib/hadoop-hdfs
sudo -u hdfs hadoop fs -mkdir /var/lib/hadoop-hdfs/cache
sudo -u hdfs hadoop fs -mkdir /var/lib/hadoop-hdfs/cache/mapred
sudo -u hdfs hadoop fs -mkdir /var/lib/hadoop-hdfs/cache/mapred/mapred
sudo -u hdfs hadoop fs -mkdir /var/lib/hadoop-hdfs/cache/mapred/mapred/staging
sudo -u hdfs hadoop fs -chmod 1777 /var/lib/hadoop-hdfs/cache/mapred/mapred/staging
sudo -u hdfs hadoop fs -chown -R mapred /var/lib/hadoop-hdfs/cache/mapred

for service in /etc/init.d/hadoop-0.20-mapreduce-*
> do
> sudo $service start
> done
```

`jps` doit maintenant nous retourner un JobTracker et un TaskTracker :

```
sudo /usr/local/java/jdk1.6.0_35/bin/jps
2099 JobTracker
1066 DataNode
1152 NameNode
2264 Jps
1248 SecondaryNameNode
2177 TaskTracker
```

Nous créons enfin le répertoire utilisateur pour nos utilisateurs Linux :

```
sudo -u hdfs hadoop fs -mkdir  /user/rudi
sudo -u hdfs hadoop fs -chown rudi /user/rudi
```

Tester l'exécution d'Hadoop

Nous allons tester Hadoop en utilisant la collection d'exemples. Elle est située dans un fichier `.jar` (une archive Java contenant des classes et d'autres ressources, utilisée pour packager une application ou des bibliothèques).

Nous allons utiliser un programme exemple situé dans le fichier `.jar` et nommé WordCount (cf. http://wiki.apache.org/hadoop/WordCount). Comme son nom l'indique, il compte le nombre

d'occurrences de chaque mot dans les fichiers contenus dans un répertoire. L'exécutable à lancer est `$HADOOP_HOME/bin/hadoop`. Utilisez la commande suivante pour le localiser :

```
whereis hadoop
```

Dans notre installation réalisée à l'aide de Cloudera, il s'agit de `/usr/bin/hadoop`. Il a été ajouté dans les alternatives, il est donc visible dans le path. Nous allons exécuter Hadoop en utilisant le compte `hdfs` créé par l'installation, et lancer le fichier `.jar` :

```
hadoop jar hadoop*examples*.jar
```

Installation manuelle

Si vous avez installé Hadoop manuellement (voir annexe), prenez l'identité de l'utilisateur `hadoop` et lancez Hadoop sous ce compte, en utilisant `sudo -u hadoop`.

Si le fichier `.jar` est trouvé dans le chemin, tout va bien, sinon vous recevrez une erreur Java `java.io.FileNotFoundException`, ou éventuellement une erreur `java.util.zip.ZipException`. Dans ce cas, cherchez l'emplacement du fichier `.jar`, par exemple avec la commande :

```
find . -iname «*example*.jar» -print
```

dans le répertoire `$HADOOP_HOME`. Pour notre exemple, le fichier et son chemin sont `/usr/share/doc/hadoop-0.20-mapreduce/examples/hadoop-examples.jar`. L'appel sera donc :

```
hadoop jar /usr/share/doc/hadoop-0.20-mapreduce/examples/hadoop-examples.jar
```

Si tout est correct, le fichier `.jar` retournera ce message :

```
An example program must be given as the first argument.
```

Nous allons travailler avec des fichiers texte récupérés du site du projet Gutenberg, qui propose des ouvrages numérisés du domaine public. Afin de nous assurer d'un corpus suffisant, nous avons pris des œuvres d'Alexandre Dumas (http://www.gutenberg.org/browse/authors/d#a492), comme *Le Comte de Monte-Cristo* ou *Le Capitaine Arena*. 28 fichiers texte au total pour environ 17 Mo. Nous les avons déposés dans notre répertoire local `~/dumas/`, puis nous allons les déplacer dans HDFS et les traiter avec Hadoop.

```
hadoop fs -mkdir /user/rudi/input
hadoop fs -put /dumas/*.txt /user/rudi/input
```

Lançons d'abord une commande pour lister tous les mots présents dans les fichiers :

```
hadoop jar /usr/share/doc/hadoop-0.20-mapreduce/examples/hadoop-examples.jar wordcount
/user/rudi/input /user/rudi/output
```

La commande lance Hadoop en spécifiant le fichier `.jar` contenant le programme Hadoop à exécuter, le programme WordCount, ainsi que le répertoire d'entrée et de sortie dans HDFS. Le

répertoire de sortie ne doit pas exister au préalable, il sera créé par le programme. Il faut donc que l'utilisateur qui lance cette commande ait les droits adéquats sur ces répertoires, c'est pourquoi nous utilisons ici `/user/rudi`, notre répertoire personnel. L'exécution est assez verbeuse. Voici le début :

```
12/11/20 18:43:50 WARN mapred.JobClient: Use GenericOptionsParser for parsing the
arguments. Applications should implement Tool for the same.
12/11/20 18:43:50 INFO input.FileInputFormat: Total input paths to process : 28
12/11/20 18:43:51 INFO mapred.JobClient: Running job: job_201211201828_0002
12/11/20 18:43:52 INFO mapred.JobClient:  map 0% reduce 0%
12/11/20 18:44:12 INFO mapred.JobClient:  map 3% reduce 0%
12/11/20 18:44:14 INFO mapred.JobClient:  map 7% reduce 0%
12/11/20 18:44:31 INFO mapred.JobClient:  map 10% reduce 0%
12/11/20 18:44:33 INFO mapred.JobClient:  map 10% reduce 2%
12/11/20 18:44:34 INFO mapred.JobClient:  map 14% reduce 2%
12/11/20 18:44:36 INFO mapred.JobClient:  map 14% reduce 4%
```

L'exécution prendrait quelques petites secondes si nous exécutions Hadoop sans HDFS sur des fichiers locaux. Elle est nettement plus lente en mode pseudo-distribué en utilisant HDFS, qui est vraiment destiné à être utilisé dans un environnement réellement distribué. Voici la dernière information produite par notre exécution :

```
Job complete: job_201211201828_0002
Counters: 32
  File System Counters
    FILE: Number of bytes read=8272152
    FILE: Number of bytes written=21081811
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
    HDFS: Number of bytes read=18066032
    HDFS: Number of bytes written=1737633
    HDFS: Number of read operations=56
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=1
  Job Counters
    Launched map tasks=28
    Launched reduce tasks=1
    Data-local map tasks=28
    Total time spent by all maps in occupied slots (ms)=384048
    Total time spent by all reduces in occupied slots (ms)=180854
    Total time spent by all maps waiting after reserving slots (ms)=0
    Total time spent by all reduces waiting after reserving slots (ms)=0
  Map-Reduce Framework
    Map input records=439824
    Map output records=2889149
    Map output bytes=29028140
    Input split bytes=3184
    Combine input records=2889149
    Combine output records=508042
```

```
    Reduce input groups=140107
    Reduce shuffle bytes=7722869
    Reduce input records=508042
    Reduce output records=140107
    Spilled Records=1051934
    CPU time spent (ms)=63220
    Physical memory (bytes) snapshot=5985443840
    Virtual memory (bytes) snapshot=27786665984
    Total committed heap usage (bytes)=4100964352
```

Vous pouvez voir les statistiques d'exécution pour les opérations de `Map` et de `Reduce`. Le répertoire destination a été créé et contient deux fichiers. Le premier est un fichier qu'on pourrait appeler sémaphore, vide, dont la présence consiste à indiquer l'état d'achèvement de la tâche. Pour notre exemple, il s'appelle `_SUCCESS`. Le second fichier contient bien sûr le résultat du programme WordCount. Il s'appelle `part-r-00000` et consiste en un simple fichier texte, dont voici un extrait.

```
Carnava,        1
Carolina        1
Carolina,       10
Carolina.       1
Caroline        49
Caroline,       32
Caroline.       11
Caroline:       2
Caroline;       2
Caroline?       2
```

Les mots trouvés sont listés avec le nombre d'occurrences. C'est une liste de paires clé-valeur exportée ici en texte, séparées par une tabulation.

Code source de l'exemple

Le code source de l'application exemple WordCount est disponible à l'adresse : http://wiki.apache.org/hadoop/WordCount. Nous reproduisons ici le code dans son intégralité car il est court et utile pour observer la simplicité de développement d'une application utilisant les fonctionnalités MapReduce d'Hadoop.

```
package org.myorg;
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
```

```
public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
 }

 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }

}
```

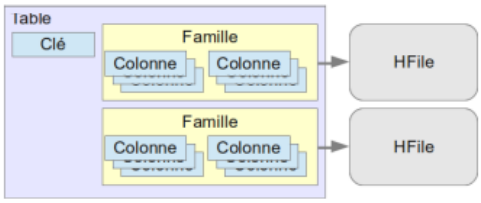
Après les lignes d'importation nécessaires des classes Java et Hadoop (`org.apache.hadoop.*`), une classe `WordCount` est créée, elle représente l'application. Elle déclare deux sous-classes statiques dérivées des classes de base de Hadoop : `Mapper` et `Reducer`. Voici la signature des classes (situées dans l'espace de noms `org.apache.hadoop.mapreduce`) :

```
Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
```

HBase

Intéressons-nous maintenant à HBase (http://hbase.apache.org/). Il s'agit d'un clone de BigTable basé sur Hadoop et donc d'un moteur distribué orienté colonnes. La figure 5-6 illustre le modèle de stockage de HBase.

Figure 5-6
Modèle de stockage de HBase



Une table organise des données par rapport à une clé, et contient des familles de colonnes, qui comportent des colonnes versionnées par un horodatage (timestamp). Voyons plus en détail chaque élément.

- La **clé** est le point d'entrée de la donnée. En HBase, elle peut être binaire. Chaque valeur de clé est bien entendu unique, et les clés sont triées par ordre lexicographique (par opposition à un ordre numérique, ainsi 2 viendra après 10, par exemple).
- Une **famille de colonnes** permet de regrouper un certain nombre de colonnes. Toutes les colonnes de la famille sont stockées ensemble sur le disque, dans un **HFile**. Il est possible de spécifier des options à la famille de colonnes, comme la compression.
- Une **colonne** contient une donnée versionnée, c'est-à-dire que chaque valeur de colonne est accompagnée d'un timestamp, qui peut être attribué par le système.
- Le **timestamp** sert à gérer la cohérence finale, mais aussi à gérer éventuellement des versions au niveau de la logique cliente. On peut indiquer que les colonnes doivent maintenir un certain nombre de versions, et les versions précédentes seront purgées automatiquement.

Le chemin pour trouver une valeur est donc le suivant :

Table → Clé → Famille → Colonne → Timestamp

Indiquer le timestamp dans la requête n'est pas nécessaire. Par défaut, vous recevrez la version la plus récente.

Architecture

HBase est pleinement conçu pour être distribué. Il se base sur Hadoop et HDFS pour distribuer son stockage. Les tables sont shardées automatiquement et séparées en régions. Cette séparation est gérée par des serveurs de régions *(RegionServer)*. Tout d'abord, la table est contenue dans une seule région. Lorsque la taille d'une région dépasse un seuil prédéfini, HBase découpe la table en deux régions, de tailles approximativement égales, sur une valeur de clé correspondant à la moitié environ de la rangée à séparer. C'est une opération rapide. Chaque région gérera donc une rangée contiguë de clés.

HBase comporte plusieurs démons. Le tableau 5-1 explique brièvement ces éléments.

Tableau 5-1. **Les démons de HBase**

Démon	Description
`RegionServer`	Démon de gestion des régions, lesquelles permettent de distribuer les tables.
`ZooKeeper`	Service centralisé de maintenance de configuration et de synchronisation pour les applications distribuées (http://zookeeper.apache.org/). C'est le pendant du Chubby de Google. Il sert à maintenir l'état des nœuds, à promouvoir un nouveau maître si le courant est tombé, etc.
`HMaster`	Le démon HBase, le moteur de bases de données lui-même. Un HMaster gère plusieurs RegionServers et centralise la gestion d'un cluster HBase.

Structure des écritures

Les données écrites sont gérées dans un WAL *(Write-Ahead Log)* et dans un buffer mémoire nommé le *memstore*. Lorsque le memstore a atteint une taille définie, il est écrit sur HDFS dans un HFile, un ensemble trié de clé-valeur sérialisé sur disque, immutable, écrit en une fois à partir du memstore.

> **Fonctionnement des suppressions**
> Si le HFile est immutable, comment les suppressions sont-elles gérées? Grâce à un mécanisme de marqueurs nommés *tombstones*. La suppression n'est pas effective immédiatement, elle sera effectuée au compactage en se référant aux marqueurs de suppression.

La réduction des différents HFile d'une famille de colonnes est effectuée par une phase de compactage. Cette structure est classique des descendants de BigTable, et nous la retrouverons pratiquement à l'identique quand nous étudierons Cassandra.

Ainsi, les régions assurent la distribution, HDFS la réplication.

Installation en mode standalone

Il n'y a pas vraiment d'intérêt à faire fonctionner HBase en mode pseudo-distribué. Nous avons déjà montré un exemple de mode pseudo-distribué pour Hadoop, le principe est le même, HBase se base sur HDFS. Si vous devez utiliser HBase en production, vous le ferez en mode pleinement distribué. Installons-le à partir des paquets Cloudera, en mode standalone :

```
sudo apt-get install hbase-master
```

L'installation doit faire démarrer le démon. Dans le cas contraire, vous pouvez le démarrer manuellement :

```
sudo service hbase-master start
```

Les choses sont simplifiées ici car en mode standalone, le serveur de région et ZooKeeper sont exécutés en interne par HMaster. En mode distribué, vous devez les configurer et les lancer indépendamment.

Résolution des problèmes d'installation

Si vous rencontrez des problèmes lors de vos premières installations d'un élément de la pile Hadoop, la première chose à faire consiste à consulter les logs. Dans la distribution Cloudera, le log de HBase se trouve dans `/var/log/hbase`. Résolution des problèmes d'installation

Si vous rencontrez des problèmes lors de vos premières installations d'un élément de la pile Hadoop, la première chose à faire consiste à consulter les logs. Dans la distribution Cloudera, le log de HBase se trouve dans `/var/log/hbase`.

Sur notre machine Ubuntu, nous avons rencontré un problème avec le fichier `hosts`. Si vous démarrez le shell comme indiqué ci-après et que les commandes que vous lancez ne répondent pas, vérifiez votre fichier `hosts`. Supprimez la référence à `localhost` sur la boucle locale, indiquez à la place le nom de la machine et ajoutez le nom `hbase` sur l'IP de l'interface :

```
127.0.0.1       nom_de_la_machine
192.165.0.27    nom_de_la_machine  hbase
```

Notez que sur Ubuntu, vous trouvez en général cette ligne :

```
127.0.1.1       nom_de_la_machine
```

Si c'est le cas, supprimez-la, conservez uniquement 127.0.0.1, et relancez le démon `hbase-master` :

```
sudo service hbase-master restart
```

Le statut du master est visible dans une interface web, par défaut sur le port 60010 (figure 5-7).

Figure 5-7
L'interface web de HBase

Mise en œuvre

Nous allons voir comment mettre en œuvre Hbase à l'aide de l'invite de commandes, et avec du code client.

Utilisation du shell

L'invite de commandes HBase est un interpréteur JRuby, qui comporte en plus des commandes propres à HBase. Pour le lancer, saisissez :

```
hbase shell
```

Les commandes supportées par le shell peuvent évidemment être obtenues via la commande `help`. `exit` permet de le quitter.

Voyons d'abord si tout est correct :

```
status
```

qui retourne :

```
1 servers, 0 dead, 0.0000 average load
```

La commande `create` permet de créer une table, à laquelle on ajoute un dictionnaire de déclarations de familles de colonnes avec l'opérateur lambda (la syntaxe de définition de hachages en Ruby). Créons notre table d'articles pour `Passerelles` :

```
create 'article', {NAME => 'auteur', VERSIONS => 3}, {NAME => 'contenu', VERSIONS =>
10}, {SPLITS => ['1000', '2000', '3000', '4000']}
```

Cette commande crée une table nommée `article`, contenant deux familles de colonnes : `auteur` et `contenu`. La famille de colonnes `auteur` conservera trois versions dans ses colonnes, et `contenu`

en gardera 10. La table est shardée manuellement sur les valeurs de clés '1000', '2000', '3000', '4000'. Cette dernière indication est totalement optionnelle, elle permet de spécifier des valeurs de partitionnement au lieu de laisser HBase décider par lui-même. Il peut être plus efficace de décider du sharding manuellement, à l'avance, cela aide à prédéfinir clairement votre système.

Vérifions que la table existe :

```
list
```

qui retourne :

```
TABLE
article
1 row(s) in 0.0100 seconds
```

Insérons maintenant notre article :

```
put 'article', '1', 'auteur:prénom', 'Annie'
put 'article', '1', 'auteur:nom', 'Brizard'
put 'article', '1', 'auteur:e-mail', 'annie.brizard@cocomail.com'
put 'article', '1', 'contenu:titre', 'pourquoi les éléphants ont-ils de grandes
oreilles?'
```

Et récupérons l'information :

```
get 'article', '1'
```

Nous obtenons :

```
COLUMN CELL
 auteur:e-mail timestamp=1353486681849, value=annie.brizard@cocomail.com
 auteur:nom timestamp=1353486653844, value=Brizard
 auteur:pr\xC3\xA9nom timestamp=1353486631132, value=Annie
 contenu:titre timestamp=1353486716515, value=pourquoi les \xC3\xA9l\xC3\xA9phants
ont-ils de grandes oreilles?
4 row(s) in 0.0400 seconds
```

Ajoutons une nouvelle version d'une colonne :

```
put 'article', '1', 'contenu:titre', 'je reviens sur mon histoire d''oreilles'
```

Et retrouvons toutes les versions de cette cellule :

```
get 'article', '1', {COLUMN => 'contenu:titre', VERSIONS => 2}
```

Nous obtenons :

```
COLUMN CELL
 contenu:titre timestamp=1353487625448, value=je reviens sur mon histoire d'oreilles
 contenu:titre timestamp=1353487561109, value=pourquoi les \xC3\xA9l\xC3\xA9phants
ont-ils de grandes oreilles?
```

```
2 row(s) in 0.0090 seconds
```

Si nous voulons supprimer la ligne :

```
deleteall 'article', '1'
```

nous pourrions être tenté d'utiliser la commande `delete`, mais elle sert à supprimer une valeur de cellule et non une ligne. Pour supprimer la table, il faut la désactiver au préalable :

```
disable 'article'
drop 'article'
```

Programmation client

Il est plus facile d'accéder à HBase à partir de Java et des classes `org.apache.hadoop.` et `hbase.client`. L'accès par un autre langage client est possible grâce à Thrift ou Avro. Nous avons vu comment installer Thrift dans le chapitre 3 consacré aux choix techniques du NoSQL. Voyons maintenant comment se connecter à HBase.

Comme nous utilisons la distribution Cloudera, nous installons le serveur Thrift pour HBase :

```
sudo apt-get install hbase-thrift
```

Nous générons d'abord les classes Thrift à partir de la définition du document HBase, que nous récupérons à l'adresse suivante : http://svn.apache.org/viewvc/hbase/trunk/hbase-server/src/main/resources/org/apache/hadoop/hbase/thrift/Hbase.thrift.

```
wget https://svn.apache.org/repos/asf/hbase/trunk/hbase-server/src/main/resources/org/
apache/hadoop/hbase/thrift/Hbase.thrift
thrift --gen py Hbase.thrift
```

Nous exécutons ensuite le script suivant :

```
#!/usr/bin/python
# coding=utf-8

import sys
sys.path.append('gen-py')

from thrift.transport.TSocket import TSocket
from thrift.transport.TTransport import TBufferedTransport
from thrift.protocol import TBinaryProtocol
from hbase import Hbase

transport = TBufferedTransport(TSocket('127.0.0.1', 9090))
transport.open()
protocol = TBinaryProtocol.TBinaryProtocol(transport)
client = Hbase.Client(protocol)

print client.getTableNames()
```

```
print client.getRow('article', '1')
```

Nous ajoutons au chemin des classes Python le répertoire dans lequel ont été générées les classes Thrift pour HBase: `sys.path.append('gen-py')`. Nous importons ces classes (`hbase`), puis nous ouvrons un transport. Nous récupérons ensuite le nom des tables présentes sur le serveur, ainsi que notre ligne d'articles, avec la clé 1.

La documentation de l'API Thrift est disponible à cette adresse : http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/thrift/doc-files/Hbase.html.

Un client Python est actuellement en développement, il s'agit de HappyBase (https://github.com/wbolster/happybase) qu'il est déjà possible d'utiliser.

Interface REST

HBase offre également une interface REST. Pour l'installer avec Coudera, utilisez la commande :

```
sudo apt-get install hbase-rest
```

Gestion des tables

Nous pouvons déclencher manuellement le compactage ou l'ajout d'une partition en utilisant l'interface web d'administration, comme illustré sur la figure 5-8.

Figure 5-8
Outils de l'interface web

Pour cela, il convient d'utiliser les commandes `compact` et `split` du shell. D'autres commandes sont disponibles pour manipuler les nœuds du cluster, tapez `help` pour les explorer:

```
help 'replication'
```

6

Le Big Data Analytique

Depuis peu de temps, tout l'environnement Hadoop a très fortement évolué. Les limitations de l'algorithme MapReduce, principalement les lenteurs de fonctionnement et l'obligation de rester en mode batch, ont été progressivement levées. Le domaine de l'analytique des grands volumes (le Big Data analytique) est devenu le sujet important du moment, avec beaucoup d'innovations à la clé récemment. Aussi nous souhaitons dans ce chapitre dresser un panorama des technologies actuelles du Big Data analytique, afin de vous aider à en comprendre les offres et les enjeux, et à faire le choix parmi la pléthore d'outils disponibles.

Présentation

Nous avons parlé d'Hadoop 2, nom de code YARN. Les nouveaux environnements de traitement distribué qui évoluent autour de YARN (ou éventuellement de son jeune concurrent Mesos) permettent de répondre à deux défis : l'analyse interactive et le traitement de larges volumes de données en temps réel. L'analyse interactive se réalise maintenant à l'aide de requêtes SQL qui semblent traditionnelles mais qui vont déclencher une analyse distribuée sur un cluster large. Le traitement en temps réel s'effectue à l'aide d'algorithmes de type DAG *(Directed Acyclic Graph)*, de traitements de flux et d'algorithmes de Machine Learning. Ces outils ont permis la naissance d'un nouveau poste informatique : celui de *data analyst* ou *data scientist*.

Dans ce chapitre, nous allons d'abord présenter une démarche Big Data analytique classique avec Sqoop et Hive, puis nous parlerons d'un point de vue plus haut niveau des solutions émergentes du Big Data analytique.

Sqoop et Hive

Nous avons déjà suffisamment parlé d'installation. Ici, nous allons plus simplement effectuer notre démonstration avec la machine virtuelle de Cloudera, Cloudera quickstart VM, dans laquelle toute la suite Cloudera communautaire (CDH) est préinstallée. Ce qui nous intéresse à ce stade, c'est de comprendre comment sont réalisés en pratique l'importation et le requêtage des données dans notre cluster Hadoop.

Nous téléchargeons donc la machine virtuelle sur http://www.cloudera.com/[8]. Nous prenons pour notre part l'image importable dans VirtualBox. Vous disposez également d'une machine virtuelle VMWare. Il vous est aussi possible d'essayer ces outils en ligne, en choisissant sur le site de Cloudera, l'option « try Cloudera Live ».

Nous importons l'image de la machine virtuelle dans VirtualBox et nous démarrons. Elle est configurée pour 4 Go de RAM. Si vous avez assez de mémoire sur votre machine hôte, augmentez cette valeur. Le système d'exploitation de la machine virtuelle est une distribution CentOS (un Red Hat serveur communautaire) avec une interface graphique gnome 2.

À l'heure où nous écrivons, la version de CDH est la 5.3. Elle inclut une installation d'HDFS, Hadoop YARN, HBase et différents outils associés, notamment Apache Spark et Impala.

L'ouverture de la machine virtuelle vous amène sur une page d'accès aux fonctionnalités incluses : http://quickstart.cloudera/. Nous allons, à l'aide de CDH et de ces exemples, faire un tour d'horizon des briques classiques d'un système de Big Data analytique.

Importer les données avec Apache Sqoop

Sqoop (http://sqoop.apache.org/) est l'outil de facto pour échanger des jeux de données entre des bases relationnelles traditionnelles et un environnement Hadoop. C'est donc un type d'outil d'ETL (Extract Transform Load) comme on en a l'habitude dans le monde de l'informatique décisionnelle. Les ETL traditionnels sont disponibles en modèle commercial : Informatica, Oracle Data Integrator ou SQL Server Integration Services, pour en nommer quelques-uns. Des outils d'ETL libres existent également, principalement Talend et Pentaho Kettle[9].

Sqoop, pour sa part, est spécialement conçu pour les allers-retours entre une base relationnelle et HDFS. Il supporte également des moteurs NoSQL en destination : Hbase, Accumulo et Cassandra. Hive, Hcatalog et Parquet sont aussi des destinations nativement implémentées[10].

Sa force est de s'appuyer comme toutes les autres briques sur un modèle distribué. C'est un projet mature, top-level Apache depuis 2012, complètement redéveloppé actuellement dans une version Sqoop2 qui n'est pas encore prête pour la production. La version stable de production au moment où nous écrivons ces lignes est la 1.4.5.

---

8. Nous évitons ici d'écrire l'URL complète, dans la mesure où elle a tendance à changer lorsque Cloudera met à jour son site. Cherchez simplement CDH (qui est à peu près l'abréviation de Cloudera Distribution of Hadoop) dans les produits sur le site.

9. D'ailleurs, Talend, une société française, propose un environnement Hadoop lié à leur outil d'ETL graphique ; le tout est nommé Talend Open Studio for Big Data (http://www.talend.com/get/bigdata).

10. Le support de Parquet notamment a été réalisé en septembre 2014 via un développement conjoint d'Intel et de Cloudera, voir http://ingest.tips/2014/11/10/parquet-support-arriving-in-sqoop/.

Sqoop est donc l'outil de choix pour ce qu'on appelle en Hadoop le processus d'ingestion *(ingestion process)*, c'est-à-dire la récupération des données opérationnelles de production pour nourrir le cluster analytique. Il peut ensuite assurer l'exportation des fichiers résultat des batchs MapReduce (les fichiers d'*output*), ou toute autre analyse vers une base relationnelle, MySQL par exemple, pour un traitement opérationnel ou l'affichage aux utilisateurs.

L'importation est réalisée avec une connexion JDBC vers la source, et vers une destination traditionnellement HDFS qui peut être de format CSV, Avro binaire ou SequenceFile. L'importation est effectuée avec des threads qui s'exécutent en parallèle, découpant par défaut la table source en segments à partir de sa clé primaire, et le résultat consistera donc en plusieurs fichiers sur HDFS.

Tables sans clé primaire

Si vous souhaitez importer une table sans clé primaire, ou dont la clé primaire est composite (composée de plusieurs colonnes), vous devez utiliser l'option `split-by` pour indiquer sur quelle colonne effectuer la segmentation. Dans ce cas, vous devez donc importer une table à la fois.

Sqoop récupère les métadonnées des tables qu'il importe et génère au passage une classe Java qui encapsule les méthodes d'accès aux structures de données importées pour MapReduce. Pour le dire autrement : Sqoop lit la structure des tables, importe les données, et génère du code Java (jar et source) avec les fonctions d'input et output pour MapReduce. Vous pouvez le faire indépendamment de l'import avec la commande `sqoop codegen`.

Mais nous n'aurons pas besoin de cela pour notre petite démonstration. Nous allons importer les tables de la base d'exemple de CDH et nous allons ensuite les utiliser pour une requête dans Hive.

L'exemple d'appel de Sqoop de Cloudera importe les tables en Avro. Nous allons être plus à la pointe de l'actualité en générant des données en Parquet, qui est maintenant une destination nativement supportée par Sqoop. Nous ouvrons un terminal dans la machine virtuelle CDH, et nous lançons la commande Sqoop suivante :

```
sqoop import-all-tables \
          -m 5 \
          --connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
          --username=retail_dba \
          --password=cloudera \
          --as-parquetfile \
          --warehouse-dir=/user/hive/warehouse > import.log
```

Dans l'ordre : nous appelons Sqoop et nous lui demandons d'exécuter la commande `import-all-tables`, qui comme son nom l'indique, importe toutes les tables de la base de données que nous indiquons dans la chaîne de connexion JDBC qui suit dans l'option `--connect`. Pour importer une seule table, la commande s'appelle simplement `import`. On peut lui indiquer une table ou une requête SQL si on souhaite filtrer les données.

Voici un exemple d'appel de Sqoop pour une table :

```
sqoop import-all-tables \
          -m 5 \
          --connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
```

```
          --table customers
…
```

L'option `-m` indique le nombre de mappers à exécuter simultanément. En fait, Sqoop utilise Hadoop pour son importation. Il va générer des fonctions map pour l'importation des données et utiliser le cluster Hadoop pour exécuter les tâches d'importation. En segmentant la source (le *split*), il peut donc s'appuyer sur le cluster Hadoop pour importer de larges volumes de données d'une façon totalement distribuée.

Tables volumineuses dans MySQL

Si vous désirez importer des tables de grande taille depuis MySQL, méfiez-vous du pilote JDBC pour MySQL. Celui-ci charge par défaut toute la table en mémoire et va remplir le stack de votre JVM. Vous avez donc de fortes chances de subir une erreur de type `OutOfMemoryException`. Utilisez la méthode `setFetchSize()` du pilote JDBC pour changer ce comportement. Vous trouverez plus de détails dans la documentation de MySQL : http://dev.mysql.com/doc/connector-j/en/connector-j-reference-implementation-notes.html.

Nous indiquons enfin que nous voulons générer des tables Parquet, et qu'elles seront situées dans le répertoire `/user/hive/warehouse` dans HDFS.

Sqoop peut également importer directement de Hive, en créant les informations de métadonnées des tables Hive dans son metastore. Cherchez la commande `--hive-import` dans la documentation de Sqoop si vous souhaitez l'utiliser.

Nous avons résumé la sortie affichée par la commande Sqoop pour ne conserver que les détails intéressants de l'importation d'une des tables de la base MySQL.

```
sqoop.Sqoop: Running Sqoop version: 1.4.5-cdh5.3.0
manager.MySQLManager: Preparing to use a MySQL streaming resultset.
tool.CodeGenTool: Beginning code generation
manager.SqlManager: Executing SQL statement: SELECT t.* FROM 'categories' AS t LIMIT 1
orm.CompilationManager: Writing jar file: /tmp/sqoop-cloudera/compile/0aa84631ee6e68f91
fb52ad6dd3b203e/categories.jar
mapreduce.ImportJobBase: Beginning import of categories
manager.SqlManager: Executing SQL statement: SELECT t.* FROM 'categories' AS t LIMIT 1
mapreduce.JobSubmitter: number of splits:5
mapreduce.JobSubmitter: Submitting tokens for job: job_1425111945531_0007
impl.YarnClientImpl: Submitted application application_1425111945531_0007
mapreduce.Job: The url to track the job: http://quickstart.cloudera:8088/proxy/
application_1425111945531_0007/
mapreduce.Job: Running job: job_1425111945531_0007
mapreduce.Job:  map 0% reduce 0%
mapreduce.Job:  map 60% reduce 0%
mapreduce.Job:  map 100% reduce 0%
mapreduce.Job: Job job_1425111945531_0007 completed successfully
mapreduce.Job: Counters: 30
        File System Counters
                FILE: Number of bytes written=660635
                HDFS: Number of bytes read=31255
```

```
                HDFS: Number of bytes written=10230
        Job Counters
                Launched map tasks=5
                Other local map tasks=5
                Total time spent by all map tasks (ms)=237299
                Total megabyte-seconds taken by all map tasks=242994176
mapreduce.ImportJobBase: Transferred 9.9902 KB in 130.3421 seconds (78.4858 bytes/sec)
mapreduce.ImportJobBase: Retrieved 58 records.
```

Vous voyez que Sqoop fait un premier appel à la table, avec l'option `LIMIT 1` du `SELECT` dans MySQL, pour récupérer les métadonnées de la table. Ensuite, il construit un mapper qu'il compile dans un fichier jar pour l'envoyer à MapReduce. Le job MapReduce démarre, affiche son état d'avancement puis ses statistiques d'exécution à la fin de son travail. Il n'y a ici qu'une phase map, l'importation ne nécessite aucune agrégation via un reducer. À la fin, Sqoop retourne des informations de taux de transfert et de nombre de lignes affectées.

On peut vérifier que les données sont bien importées en affichant la liste des sous-répertoires de notre répertoire de destination :

```
hadoop fs -ls /user/hive/warehouse
```

Mais on va le voir graphiquement avec Hue.

Si vous avez réalisé l'import et que vous souhaitez le recommencer, vous obtiendrez une erreur. En effet, Sqoop n'écrase pas par défaut les données. Vous pouvez lui demander de le faire à l'aide de l'option `--delete-target-dir`, ou vous pouvez supprimer le répertoire au préalable :

```
hadoop fs -rm -f -r /user/hive/warehouse
```

Si vous souhaitez réaliser des importations incrémentales, vous pouvez importer dans des répertoires horodatés si vous travaillez avec des fichiers plats, ou vous pouvez utiliser l'option `--append` pour demander à Sqoop d'ajouter les données.

Voici quelques arguments intéressants de l'importation.

Tableau 6-1 Arguments de Sqoop

Argument	Description
`--append`	Ajoute à un jeu de données déjà existant en HDFS. Pour les formats de fichiers, crée des nouveaux fichiers incrémentiels avec un autre nom.
`--as-avrodatafile`	Importe en Avro
`--as-sequencefile`	Importe en SequenceFiles
`--as-textfile`	Importe en fichiers texte (c'est le format par défaut)
`--boundary-query <statement>`	Requête pour indiquer les segments de splits
`--columns <col,col,col…>`	Si présent, n'importe que les colonnes mentionnées
`--delete-target-dir`	Supprime le répertoire de destination
`--fetch-size <n>`	Nombre de lignes à récupérer en une fois
`-m,--num-mappers <n>`	Nombre de tâches de mapper à exécuter en parallèle

`-e,--query <statement>`	Importe le résultat de la requête `SELECT`
`--split-by <column-name>`	Colonne sur laquelle effectuer le split. Par défaut, la clé primaire. Si la clé primaire est composite (placée sur plusieurs colonnes), vous devez indiquer cette option et choisir une seule colonne.
`--table <table-name>`	Nom de la table pour la commande import
`--target-dir <dir>`	Destination HDFS pour les fichiers
`--warehouse-dir <dir>`	Destination HDFS pour le répertoire parent de tables, selon le format d'importation
`--where <where clause>`	Clause `WHERE` à appliquer sur la source pour filtrer les données à importer
`-z,--compress`	Active la compression en destination
`--compression-codec <c>`	Choix du codec de compression (le défaut est gzip, les options correspondent aux codecs disponibles dans Hadoop)
`--null-string <null-string>`	La chaîne à écrire en destination en cas de `NULL` dans la cellule source.
`--null-non-string <null-string>`	La chaîne à écrire en destination en cas de `NULL` dans la cellule source, si elle n'est pas de type string.

Importer les tables dans Hive

Apache Hive (https://hive.apache.org/) est une surcouche analytique à Hadoop qui offre une couche d'abstration aux données HDFS sous forme d'un modèle tabulaire et relationnel, et un langage de requête SQL nommé HiveQL, dont l'objectif est de s'approcher du support de la norme SQL, et qui implémente entre autres des jointures. Le travail de Hive, c'est de transformer une requête HiveQL en un job MapReduce, de l'exécuter sur le cluster Hadoop et de récupérer les résultats sous forme de jeu de résultat tabulaire. C'est un projet né au sein de Facebook (la naissance officielle de Hive date d'août 2009, à la publication d'un *white paper* de l'équipe *Data Infrastructure* de Facebook). En février 2015, le comité de gestion de projet Apache de Hive (la fondation Apache définit des PMC – *Project Management Committee* – par projet, voir http://www.apache.org/foundation/how-it-works.html) a voté la sortie de Hive 1.0, qui correspond à l'ancienne version 0.14.1. C'est un signe pour indiquer les progrès importants effectués sur l'architecture de Hive, et le langage HiveQL.

Pour utiliser et requêter les données avec Hive, vous devez les déclarer dans Hive comme des tables, ce qui aura effet de les rendre manipulables, à travers l'abstraction des données que fournit Hive et qui s'appelle le metastore. Le metastore est un service qui donne aux clients les métadonnées des tables déclarées dans Hive et qui persiste ces métadonnées dans une base de données Apache Derby (Derby est un SGBDR libre développé en Java qui peut fonctionner comme client-serveur ou moteur *embedded*: http://db.apache.org/derby/). Par défaut, le metastore est local à la machine Hive, mais vous pouvez partager les métadonnées entre machines en installant Derby en client-serveur (https://cwiki.apache.org/confluence/display/Hive/HiveDerbyServerMode).

Pour déclarer nos tables dans le metastore, nous allons nous connecter au client interactif Hive dans notre terminal, et lancer une commande `CREATE TABLE`.

```
hive
```

Nous allons importer seulement une table, `customers`, pour donner un exemple minimal. La procédure pour l'importation des autres tables coule de source, et à plus forte raison si vous utilisez la capacité de Sqoop d'importer directement dans Hive : le `CREATE TABLE` ci-après ne sera plus pour vous que figure d'exemple.

```
CREATE EXTERNAL TABLE customers (
    customer_id int,
    customer_fname varchar(45),
    customer_lname varchar(45),
    customer_email varchar(45),
    customer_password varchar(45),
    customer_street varchar(255),
    customer_city varchar(45),
    customer_state varchar(45),
    customer_zipcode varchar(45)
)
STORED AS PARQUET
LOCATION 'hdfs:///user/hive/warehouse/customers';
```

Il faut l'avouer, nous avons choisi ici la facilité : seulement des `varchar`. En effet, attention aux types de données si vous voulez travailler avec Parquet et Hive. Le support de Parquet pour Hive étant relativement récent, tous les types de données de Hive ne sont par reconnus dans le stockage Parquet pour Hive, qui s'appelle PaquetSerDe (en Hive, un SerDe est une interface de stockage : sérialiseur, désérialiseur).

La tâche suivante vous donne un état des lieux : https://issues.apache.org/jira/browse/HIVE-6384. À l'heure où nous écrivons ces lignes (Hive 0.13), le type date n'est supporté que sous forme de patch. Nous utilisons donc ici le type timestamp.

Nous avons utilisé la syntaxe `STORED AS PARQUET` qui est nouvelle en Hive 0.13, qui est la première version à avoir un support natif de Parquet. Sur une version antérieure (à partir de Hive 0.10), vous pouvez utiliser la syntaxe suivante :

```
CREATE EXTERNAL TABLE customers (
    ...
)
ROW FORMAT SERDE 'parquet.hive.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT «parquet.hive.DeprecatedParquetInputFormat»
OUTPUTFORMAT «parquet.hive.DeprecatedParquetOutputFormat»
LOCATION 'hdfs:///user/hive/warehouse/customers';
```

Ne vous inquiétez pas pour la mention `Deprecated` que vous pouvez lire dans les formats d'import et d'export, les classes ont été nommées ainsi pour indiquer qu'elles utilisent le vieil API `org.apache.hadoop.mapred` dans Hadoop, au lieu du plus récent `org.apache.hadoop.mapreduce`.

Après création de la table, il nous suffit de vérifier que les données sont interrogeables :

```
select * from customers limit 1;
>OK
>9949   Ashley   Smith    XXXXXXXXX        XXXXXXXXX        6631 Pleasant Zephyr Route
     El Paso    TX      79927
```

```
Time taken: 0.081 seconds, Fetched: 1 row(s)
```

Pour une simple requête comme celle-ci, Hive n'a pas besoin de créer un job MapReduce. Il choisira de le faire si nous lui demandons quelque chose de plus compliqué.

Gérer les données avec Apache Hue

Hue (http://gethue.com/) est l'abréviation de Hadoop User Experience. C'est une interface web qui donne accès aux données stockées dans HDFS. Sur CDH, Hue est configuré pour répondre sur le port 8888. Il centralise l'accès aux informations de HDFS, Hive et Impala.

Hue vous permet de visualiser simplement l'arborescence HDFS (figure 6-1) et le metastore de Hive (figure 6-2).

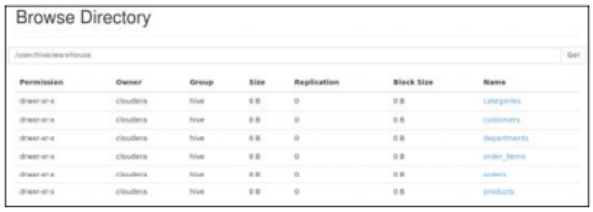


Figure 6-1
Arborescence HDFS dans Hue

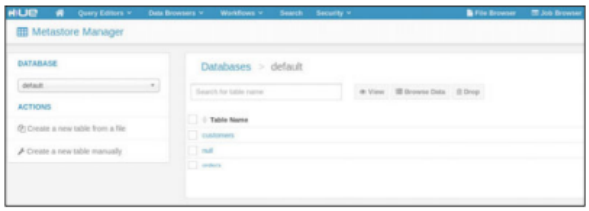
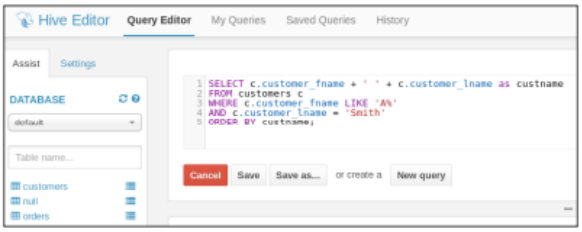


Figure 6-2
Hive metastore dans Hue

Nous allons utiliser Hue pour saisir une requête Hive comportant un filtre dans la clause `WHERE`, et observer comment l'exécution est menée.

Nous allons sur l'éditeur de requête *(Query Editor)* et nous créons une nouvelle requête Hive (figure 6-3).

Figure 6-3
Editeur de requête Hive dans Hue



Notre requête :

```
SELECT c.customer_fname + ' ' + c.customer_lname as custname
FROM customers c
WHERE c.customer_fname LIKE 'A%'
AND c.customer_lname = 'Smith'
ORDER BY custname;
```

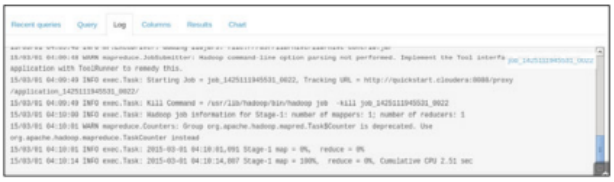
est une pure requête SQL utilisant une syntaxe définie dans la norme SQL-92, notamment pour l'utilisation du `LIKE` dans la clause `WHERE`, et la définition des alias. Il y a naturellement quelques spécificités à Hive. Vous voyez par exemple que nous réutilisons l'alias défini dans le `SELECT` comme critère de tri dans l'`ORDER BY`. C'est bien sûr parfaitement valide, car dans l'ordre d'exécution logique d'un ordre SQL, l'`ORDER BY` s'évalue après le `SELECT` et peut réutiliser les alias définis auparavant. Mais dans un SGBDR, l'`ORDER BY` peut aussi contenir des critères qui ne sont pas exprimés dans le `SELECT` : c'est ce qu'on appelle un critère externe. Dans HiveQL, ce n'est pas possible : vous devez indiquer une colonne ou un alias tel qu'il est présent dans le `SELECT`.

Support de la norme SQL

Nous l'avons dit, l'objectif de HiveQL est le support complet de la norme SQL, spécifiquement SQL-92, et dans l'idéal SQL-2011. Hive inclut notamment tous les types de jointures (internes, externes et croisées) mais seulement en équijointures (comparaison avec égalité), sous-requêtes, fonctions de fenêtrage (*windowing functions*) et fonctions utilisateurs (UDF: *user defined functions*) écrites en Java.

L'exécution de cette requête va entraîner la création d'une fonction de mapper qui sera exécutée par MapReduce. Hue affiche un log en temps réel (figure 6-4) et donne un lien sur le job MapReduce en exécution.

Figure 6-4
Journal d'exécution du job MapReduce



Le lien nous envoie sur le job browser, qui affiche également un journal des jobs exécutés et terminés (figure 6-5).

Figure 6-5
Job log dans Hue

Stinger

Hortonworks et d'autres acteurs ont lancé l'initiative Stinger (http://hortonworks.com/labs/stinger/) dont l'objectif est de rendre Hive pratiquement interactif en temps réel, d'y ajouter la possibilité de modifier les données avec une garantie ACID de la transaction, un moteur d'exécution qui vectorise le traitement (nous avons parlé des vecteurs dans notre section sur le format ORC File, page 123), des non-équijointures et des vues matérialisées.

Apache Spark

Les intiatives, les projets, les frameworks autour du Big Data analytique poussent en ce moment comme des champignons. Pour clore ce chapitre, nous allons évoquer un projet important nommé Spark.

Apache Spark est l'un des projets du Big Data analytique les plus intéressants. Il est constitué de plusieurs couches très bien intégrées qui offrent tout ce dont vous avez besoin pour dans un environnement de Big data complet. À l'heure où nous rédigeons cet ouvrage, la dernière version est la 1.2.1, sortie en février 2015. Spark a été développé à l'origine par l'université de Californie à Berkeley, puis transmis à la fondation Apache. Il est maintenant développé par environ 500 contributeurs, ce qui en fait le projet le plus actif de la fondation.

Spark (https://spark.apache.org/) est un environnement complet de stockage et de traitement distribué. Pendant quelques années, faire du Big Data analytique signifiait installer différentes couches, par exemple HDFS, HBase et Hadoop, ensuite Hive pour les requêtes. Le but de Spark est de fournir tout ce qui vous est utile dans le même outil, lequel prend en charge l'intégration de tous les modules nécessaires.

Il existe un autre environnement de ce type appartenant également à la fondation Apache, nommé Apache Tez. C'est une implémentation concurrente de Spark, elle aussi libre, dont l'équipe est issue de plusieurs entreprises. Mais la plupart des contributeurs proviennent d'une société spécialisée dans la distribution et le support de solutions Hadoop : Hortonworks. Spark en revanche est principalement distribué par Cloudera. On peut donc y voir deux produits concurrents des deux

plus importantes entreprises du monde Hadoop. Dans les faits, à l'heure actuelle, Spark est un projet plus mature, plus actif et plus riche que Tez. Une société a été fondée par les développeurs de Spark, nommée Databricks (https://databricks.com/), qui offre un environnement cloud autour de Spark, ainsi que des formations et un processus de certification.

Caractéristiques

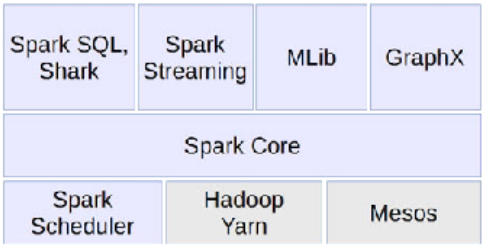
Avant de présenter l'architecture de Spark, précisons déjà quelque chose d'intéressant : il est très rapide. C'était l'un des objectifs de son développement : corriger les lenteurs d'Hadoop qui sont dues à son architecture, et notamment au fait que l'implémentation d'Hadoop MapReduce travaillait directement avec les fichiers sur le disque en HDFS. Rien n'est prévu dans MapReduce pour maintenir un cache en mémoire. Spark au contraire va charger les données en mémoire vive de façon à pouvoir optimiser de multiples traitements. Il est typiquement de 10 à 100 fois plus rapide que Hadoop MapReduce. Il existe un concours annuel de performances des environnements de Big Data, que vous trouvez à l'adresse http://sortbenchmark.org/. Le but est de trier le plus rapidement possible un jeu de données d'un volume de 100 To. Il y a plusieurs métriques et plusieurs jeux de données, dont vous trouvez les références sur le site, mais simplifions ici et conservons la métrique la plus importante sur le premier jeu de données utilisé. En 2013, le gagnant était un cluster Hadoop de Yahoo! qui est parvenu à trier 102.5 To en 4 328 secondes, sur un cluster de 2 100 nœuds de machines avec 64 Go de mémoire. Cela fait un rythme de tri de 1,42 To à la minute.

En 2014, il y a deux gagnants ex aequo : un système nommé TritonSort développé par l'université de Californie à San Diego, et Spark. Spark a trié 100 To en 1 406 secondes sur un cluster Amazon ec2 de 207 noeuds de 244 Go de mémoire et disque SSD. Le test a été réalisé par Databricks. Cela équivaut à un rythme de 4,27 To par minute, c'est-à-dire trois fois mieux qu'Hadoop avec 10 fois moins de machines (mais beaucoup plus de mémoire vive). Grâce à des environnements comme Spark, le Big Data analytique est en train de réaliser peu à peu la promesse d'un traitement volumineux en temps réel.

Architecture

Le cœur de Spark est un module nommé Spark Core. Il offre les fonctionnalités de base, à savoir la distribution et la planification des tâches, ainsi que les primitives d'entrées-sorties. Spark Core utilise pour ses entrées-sorties une abstraction nommée RDD, pour Resilient Distributed Datasets (jeux de données distribués résilients). Le nom évoque le fait que les données sont naturellement distribuées, et répliquées de façon à pouvoir survivre aux défaillances matérielles des nœuds. Cette abstraction permet à Spark Core de travailler de manière identique avec n'importe quel stockage physique sous-jacent. Cette couche offre donc un niveau intermédiaire qui permet d'allier au niveau bas n'importe quel type de stockage et différents environnements de distribution des tâches comme Hadoop YARN ou Mesos, et au niveau haut, plusieurs algorithmes ou environnements de traitement qui se basent sur Core pour l'accès à la planification et aux données. Sont actuellement disponibles avec Spark quatre formes de traitement : des moteurs d'exécution SQL pour le requêtage interactif, le module Spark Streaming pour le traitement en temps réel des flux, MLib pour les algorithmes d'apprentissage sur les données (le Machine Learning) et le module GraphX pour le traitement des graphes distribués.

Figure 6-6
Architecture de Spark



7

CouchDB et Couchbase Server

CouchDB est un système de gestion de bases de données destiné principalement aux applications web, en raison de son support des protocoles propres au Web. Les documents au format JSON sont manipulables en JavaScript à travers une interface HTTP.

CouchDB offre une grande richesse fonctionnelle au niveau du serveur, avec la possibilité de définir des filtres et des vues dans le langage de votre choix – habituellement JavaScript – avec un support de MapReduce, et qui génère des index pour offrir de meilleures performances.

Cela dit, CouchDB souffre de plusieurs défauts qui le rendent peu apte à soutenir des applications de production exigeantes. En revanche, Couchbase Server, qui est une évolution de Membase et de CouchDB, est plus adapté aux besoins de production. Nous allons étudier ces deux moteurs dans ce chapitre.

Présentation de CouchDB

CouchDB a été créé par Damien Katz, un ancien développeur de Lotus Notes chez IBM. Il le développa pendant deux ans, avant de le libérer sous Licence Apache. Son but était de faire de CouchDB un moteur de bases de données adapté à Internet. En 2011, Damien Katz a annoncé qu'il allait se concentrer sur le développement de CouchBase Server, en intégrant la société Couchbase. Il en fut le *Chief Technical Officer* jusqu'en août 2013.

Caractéristiques

CouchDB est développé en Erlang, un langage fonctionnel particulièrement adapté à l'informatique distribuée et créé par la société Ericsson (Erlang est une contraction de *Ericsson Language*). Ce langage comporte nativement des fonctionnalités de distribution des processus. CouchDB est un moteur de bases de données orienté documents, ce qui signifie qu'il stocke et manipule des structures JSON (comme MongoDB que nous étudierons au chapitre 8). Son objectif est d'être simple et intuitif à utiliser (d'où son nom relaxant de CouchDB), mais aussi adapté pour le Web. À ce titre, il offre au programmeur une interface REST qui le rend accessible de partout et avec n'importe quel langage, une gestion de la montée en charge aussi automatique que possible, et des fonctionnalités natives de gestion des données distribuées. Souvenons-nous de ces informations : Erlang, JSON, REST. Nous en retrouverons l'héritage dans Couchbase Server.

La cohérence des données est assurée par une gestion de versions de type MVCC (*Multiversion Concurrency Control*), c'est-à-dire un verrouillage optimiste qui ne bloque pas la donnée sur le serveur lorsqu'elle est en modification chez un client, mais qui s'assure au moment de la mise à jour qu'elle n'a pas été modifiée sur le serveur entre-temps.

Les vues permettent, à travers des fonctions procédurales qui s'appliquent aux données et une implémentation transparente et purement locale de l'algorithme MapReduce, d'effectuer des recherches sur des documents dont la structure n'est pas homogène à l'aide de fonctions map et reduce. Les vues permettent donc de travailler élégamment sur des données sans schéma prédéfini.

Mise en œuvre de CouchDB

Installation

Les fichiers d'installation de CouchDB peuvent être téléchargés sur le site qui lui est dédié : http://couchdb.apache.org/. À l'heure où nous écrivons ces lignes, la dernière version stable est la 1.6.1. Une version 2 est en développement. CouchDB est disponible dans les dépôts Debian/Ubuntu, en général avec au moins une version de retard. Les moteurs NoSQL évoluant vite, nous évitons en général d'installer d'anciennes versions. En d'autres termes, vous l'avez compris, nous allons effectuer une installation manuelle, ou plutôt semi-manuelle, car un outil est tout de même disponible pour nous simplifier le travail. En effet, nous allons utiliser le script `build-couchdb`, écrit en Ruby et disponible à l'adresse suivante sur le site Github : https://github.com/iriscouch/build-couchdb.

Compilation de CouchDB

Si vous souhaitez compiler CouchDB de façon entièrement manuelle, consultez la marche à suivre indiquée sur la page suivante : http://wiki.apache.org/couchdb/Installing_on_Ubuntu.

La première chose à faire est d'installer quelques dépendances nécessaires au script `build-couchdb` et CouchDB.

```
sudo apt-get install help2man make gcc zlib1g-dev libssl-dev rake help2man git
```

Nous téléchargeons ensuite `build-couchdb` depuis Github.

```
cd ~/downloads
git clone git://github.com/iriscouch/build-

cd build-couchdb
git submodule init
git submodule update
```

La commande `git submodule` permet de télécharger automatiquement les dépendances nécessaires pour compiler CouchDB. Lorsque le téléchargement est terminé, il suffit de lancer `rake`, l'outil de `make` de Ruby. `rake` va inscrire le chemin de destination de la compilation un peu partout dans les scripts de lancement de CouchDB. Pour placer CouchDB compilé directement au bon endroit, nous indiquons à `rake` le répertoire d'installation. Nous lançons la commande en `sudo`, de manière à donner les droits nécessaires à l'utilisateur courant sur notre répertoire de destination, à savoir `/usr/local/`.

```
sudo rake install=/usr/local/couchdb
```

`rake` va exécuter le `rakefile` livré avec `build-couchdb`, télécharger les dépendances nécessaires à CouchDB et le compiler. Cela peut prendre un certain temps. Vous trouverez ensuite le résultat de la compilation dans le répertoire `build/`. Vérifions d'abord que CouchDB est bien compilé et que nous avons la dernière version :

```
/usr/local/couchdb/bin/couchdb -V
```

Nous obtenons :

```
couchdb - Apache CouchDB 1.6.1
```

et la mention de la licence. Tout va bien.

Afin d'éviter d'exécuter CouchDB en tant que `root`, nous créons un utilisateur nommé `couchdb` et nous lui donnons les droits sur le répertoire de CouchDB compilé.

```
sudo adduser couchdb
sudo chown -R couchdb /usr/local/couchdb
```

Nous lançons ensuite CouchDB :

```
su couchdb
nohup /usr/local/couchdb/bin/couchdb &
```

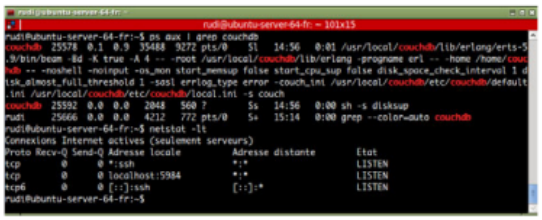
`nohup` et `&` permettent de lancer le processus en tâche de fond et de le laisser actif même si l'utilisateur ferme sa session.

Nous pouvons maintenant vérifier si CouchDB est lancé, par exemple en trouvant son processus, ou en contrôlant s'il écoute sur son port par défaut, le port TCP 5984 :

```
ps -aux | grep couchdb
netstat -lt
```

La commande `ps` retourne la liste des processus, et nous filtrons avec `grep`. La commande `netstat -lt` affiche les connexions réseaux ouvertes en écoute (`-l`) en protocole TCP/IP (`-t`). Le résultat est présenté sur la figure 7-1.

Figure 7-1
Résultat de netstat : écoute en local



CouchDB est bien lancé et écoute bien sur le port 5984. En revanche, on peut voir qu'il écoute seulement sur la boucle locale (`localhost`). Nous remarquons aussi, dans le résultat de la première commande, que les fichiers de configuration définis se trouvent aux emplacements `/usr/local/couchdb/etc/couchdb/default.ini` et `/usr/local/couchdb/etc/couchdb/local.ini`. `Default.ini` contient la configuration de CouchDB, mais vous ne devriez pas modifier ce fichier, car il sera écrasé lors d'une mise à jour et tous les changements seront alors perdus. Utilisez plutôt le fichier `local.ini`, qui ne sera pas remplacé en cas de mise à jour et dont les paramètres ont priorité sur ceux de `default.ini` (puisqu'il est listé en second dans les listes des fichiers de configuration de CouchDB). C'est donc dans `local.ini` que nous allons modifier la configuration de la façon suivante :

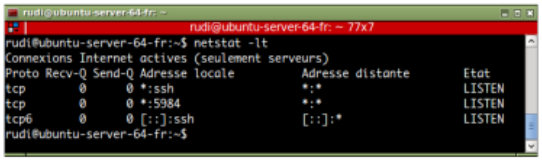
```
[httpd]
bind_address = 0.0.0.0
```

Ensuite, nous tuons CouchDB et nous relançons le processus :

```
ps -U couchdb -o pid= | sudo xargs kill -9
su couchdb
nohup /usr/local/couchdb/bin/couchdb &
```

La figure 7-2 montre le nouveau résultat du `netstat`.

Figure 7-2
Résultat de netstat : écoute sur toutes les interfaces



Nous pouvons ensuite tester l'interface de CouchDB à l'aide d'un client web ou de l'utilitaire en ligne de commandes curl. Sur la machine où CouchDB est installé, cet appel :

```
curl http://localhost:5984/
```

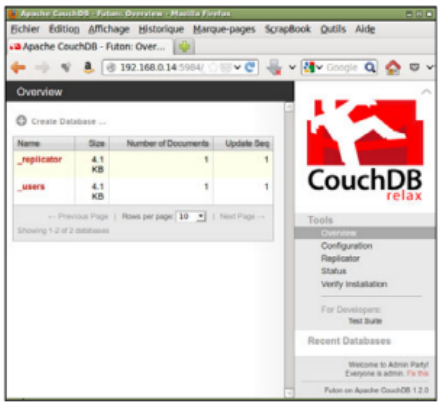
retourne un simple document :

```
{«couchdb»:»Welcome»,»version»:»1.6.1»}
```

Utilisation de Futon

Futon est l'outil web de gestion de CouchDB. Pour le trouver, appelez le chemin `_utils` de l'emplacement web de CouchDB. La figure 7-3 représente la page d'accueil du site de Futon appelée sur notre serveur http://nosql-server:5984/_utils/.

Figure 7-3
Futon



Pour vous assurer que votre installation de CouchDB est totalement fonctionnelle, rendez-vous dans le menu Tools et cliquez sur Verify Installation qui permet une validation simple. Vous pouvez également utiliser les scripts JavaScript de test livrés par Futon dans le menu For Developers: Test Suite. Lancez le test Basic.

Nous pouvons maintenant essayer de créer une base de données à l'aide de Futon. Nous créons d'abord la base de données en cliquant sur le lien Create Database situé en haut à gauche dans l'affichage de vue d'ensemble de Futon. Nous la nommons `passerelles`, tout en minuscules, car une base de données CouchDB ne peut pas contenir de majuscules. Nous sommes ensuite dirigés vers la base de données dans laquelle nous pouvons ajouter des documents, à l'aide du lien New Document. Nous voulons ajouter notre entrée de blog pour notre site Passerelles. L'interface de Futon nous propose d'ajouter le document champ par champ.

Utilisation de l'API REST

Voici quelques exemples d'appel à CouchDB à l'aide de curl (pour plus d'informations sur curl, voir la section «curl» du chapitre 3, page 67). La commande suivante crée la base de données `passerelles`.

```
curl -X PUT http://localhost:5984/passerelles
```

Nous insérons ensuite un petit document dans la base :

```
curl -X PUT http://localhost:5984/passerelles/1 -d '
{
«auteur»:
{
    «prénom»:»Annie»,
    «nom»:»Brizard»,
    «e-mail»:»annie.brizard@cocomail.com»
},
«titre»:»pourquoi les éléphants ont-ils de grandes oreilles?»
}'
```

et nous le retrouvons :

```
curl -X GET http://localhost:5984/passerelles/1
```

Le résultat est illustré sur la figure 7-4.

Figure 7-4
Appel REST avec curl

Afin de gérer la consistance des modifications de données, CouchDB maintient les révisions successives des mises à jour sur un document. Nous pouvons voir sur la figure 6-4 l'identifiant de révision dans la clé `_rev`. On peut effectuer une modification du document en refaisant un `PUT` sur la même URL de document et en indiquant le numéro de la dernière révision :

```
curl -X PUT http://localhost:5984/passerelles/1 -d '
{
«_rev»: «1-dc5def75ffe9db45c5c3de4d21ebabb8»,
«auteur»:
{
    «prénom»:»Annie»,
    «nom»:»Brizard»,
    «e-mail»:»annie.brizard@cocomail.com»
},
«titre»:»je change mon titre, il était trop bizarre»
}'
```

Ce qui retourne un nouveau numéro de révision :

```
{"ok":true,"id":"1","rev":"2-1a97e3eb387fd93cf6e1abb2a1aed56b"}
```

Numéro de révision

Le numéro de révision est un hash en MD5 de la modification, précédé d'un numéro indiquant le nombre de fois où le document a été mis à jour.

Nous avons vu dans le chapitre 3 page 89 que l'obligation d'envoyer un numéro de révision permet de gérer un mode de verrouillage optimiste.

Si vous oubliez le numéro de révision, ou si vous indiquez autre chose que le dernier numéro de révision, vous obtiendrez l'erreur suivante :

```
{"error":"conflict","reason":"Document update conflict."}
```

Prenons exemple sur la suite d'appels suivante, où sont indiquées les réponses de CouchDB :

```
curl -X PUT http://localhost:5984/passerelles/2 -d '{}'
{"ok":true,"id":"2","rev":"1-967a00dff5e02add41819138abb3284d"}
curl -X PUT http://localhost:5984/passerelles/2 -d '{}'
{"error":"conflict","reason":"Document update conflict."}
curl -X PUT http://localhost:5984/passerelles/2 -d '{"_rev":"1-967a00dff5e02add4181913
8abb3284d"}'
{"ok":true,"id":"2","rev":"2-7051cbe5c8faecd085a3fa619e6e6337"}
curl -X PUT http://localhost:5984/passerelles/2 -d '{"_rev":"1-967a00dff5e02add4181913
8abb3284d"}'
{"error":"conflict","reason":"Document update conflict."}
```

Nous avons essayé d'envoyer deux fois un document avec le même numéro de révision. La seconde fois, nous avons obtenu une erreur : le numéro de révision n'était plus le numéro actuel, il y a donc eu un conflit.

L'API de CouchDB est riche en fonctionnalités. Pour vous donner un exemple, suivons notre trace de révisions. Afin d'obtenir plus d'informations sur les révisions de notre document, nous demandons d'ajouter dans le document en retour le champ spécial `_revs_info` :

```
curl -X GET http://localhost:5984/passerelles/1?revs_info=true
```

Ce qui retourne le résultat suivant :

```
{
   «_id»:»1»,
   «_rev»:»2-1a97e3eb387fd93cf6e1abb2a1aed56b»,
   «auteur»:{
      «prénom»:»Annie»,
      «nom»:»Brizard»,
      «e-mail»:»annie.brizard@cocomail.com»
   },
   «titre»:»je change mon titre, il était trop bizarre»,
   "_revs_info":[
      {
         "rev":"2-1a97e3eb387fd93cf6e1abb2a1aed56b",
         "status":"available"
```

```
      },
      {
         «rev»:»1-dc5def75ffe9db45c5c3de4d21ebabb8»,
         «status»:»available»
      }
   ]
}
```

Nous aurions aussi pu demander la liste des numéros de révision du document sans plus de détail:

```
curl -X GET http://localhost:5984/passerelles/1?revs=true
```

Nous aurions alors obtenu le résultat suivant:

```
"_revisions":{"start":2,"ids":["1a97e3eb387fd93cf6e1abb2a1aed56b","dc5def75ffe9db45c5c
3de4d21ebabb8"]}
```

Vous avez peut-être noté que les `id` de révision diffèrent entre les deux appels. Ceci vient du fait que `_revs_info` agrège un numéro de révision avec l'ID de révision, de façon à afficher une séquence.

Nous pouvons ensuite requêter une révision particulière, par exemple:

```
curl -X GET http://localhost:5984/passerelles/1?rev=1-dc5def75ffe9db45c5c3de4d21ebabb8
```

qui retournera le titre «pourquoi les éléphants ont-ils de grandes oreilles?»

Vous pouvez obtenir toutes les révisions ou certaines révisions d'un document avec les syntaxes suivantes:

```
# toutes les révisions
curl -X GET http://localhost:5984/passerelles/1?open_revs=all
curl --globoff -X GET http://localhost:5984/passerelles/1?open_revs=["1-dc5def75ffe9db
45c5c3de4d21ebabb8»,»2-1a97e3eb387fd93cf6e1abb2a1aed56b»]
```

`_id` et `_rev` sont des champs particuliers.

Utilisation de l'API REST dans le code client

Dans la prochaine section, nous allons découvrir quelques fonctionnalités du serveur CouchDB. Mais avant cela, nous devons créer une base CouchDB avec un peu de matière. À l'aide d'un script Python, nous allons prendre un flux RSS et stocker les articles de ce flux dans une base de données CouchDB. Avant de créer le code, nous devons importer deux bibliothèques dans notre environnement Python: `feedparser`, pour manipuler le flux RSS, et `requests`, pour appeler simplement l'API REST depuis notre code.

```
~/python_env/nosql/bin/pip install feedparser
~/python_env/nosql/bin/pip install requests
```

Nous allons convertir les articles de flux en documents JSON. Nous rencontrons ici un problème, car la bibliothèque `feedparser` retourne certains éléments du flux dans des types de données incompatibles avec JSON, notamment la date de l'article qui est de type Python `time.asctime`. Pour résoudre ce problème, nous créons un module nommé `customserializer.py` que nous plaçons dans le même répertoire que le script que nous allons créer. Ce module sera utilisé par la méthode `dumps` de la bibliothèque `json` pour prendre en charge les types qui posent problème. Nous avons récupéré le code suivant du livre *Dive Into Python 3*, disponible en ligne à l'adresse suivante : http://getpython3.com/diveintopython3/serializing.html.

```
#!/usr/bin/python
# coding=utf-8

import time

def to_json(python_object):
    if isinstance(python_object, time.struct_time):
        return {'__class__': 'time.asctime',
                '__value__': time.asctime(python_object)}
    if isinstance(python_object, bytes):
        return {'__class__': 'bytes',
                '__value__': list(python_object)}
    raise TypeError(repr(python_object) + ' is not JSON serializable')
```

Ensuite, nous pouvons créer notre script, dont le code est le suivant :

```
#!/usr/bin/python
# coding=utf-8

import json
import requests
import customserializer
from feedparser import parse

url = 'http://127.0.0.1:5984/actualitte'
i = 1

# on supprime la base au cas où
print requests.delete(url)
# on la crée
print requests.put(url)

from feedparser import parse
myfeed = parse("http://www.actualitte.com/flux-rss-news.xml")

print myfeed.feed.title

for item in myfeed.entries:
        j = json.dumps(item, default=customserializer.to_json)
        cle = "%05d"% i
        print requests.put(url+'/'+cle, data=j)
```

```
        i += 1
```

Après avoir importé les modules nécessaires, nous définissons notre URL REST à `http://127.0.0.1:5984/actualitte`. `actualitte` correspond au nom du site dont nous récupérons le flux, nous en faisons donc le nom de notre base de données.

ActuaLitté

Le site ActuaLitté (http://www.actualitte.com/) est un portail dédié à l'actualité littéraire. Nous l'avons choisi simplement parce que son flux RSS est très régulièrement mis à jour.

Nous utilisons le module `requests` pour nos appels REST : `requests.delete(url)` nous permet de supprimer la base de données si elle existe déjà afin de recommencer notre test depuis le début (sans base donc). Nous effectuons ensuite un `PUT` pour (re)créer la base. À l'aide de `feedparser`, nous récupérons le flux et nous bouclons à l'intérieur des articles. Pour chaque article, nous utilisons la méthode `json.dumps` pour convertir l'objet Python en une structure JSON, en passant par notre `customserializer` pour gérer les types incompatibles. Nous prenons soin également de générer une clé pour chaque entrée, avec un compteur incrémenté dans la boucle et que nous formatons sur cinq positions (par exemple `'00001'`). Comme le stockage en B-Tree de CouchDB est réalisé dans l'ordre alphabétique des clés, nous nous assurons ainsi de bien insérer nos articles dans l'ordre, principalement pour rendre les résultats de notre exemple bien lisibles. Enfin, nous lançons un `PUT` à l'aide de `requests.put` en passant notre JSON en paramètre `data` (la documentation sur le fonctionnement de `requests.put` est disponible à cette adresse : http://docs.python-requests.org/en/latest/api/?highlight=put#requests.put).

Nous nommons notre script `get_actualitte.py` et nous l'exécutons :

```
~/python_env/nosql/bin/python get_actualitte.py
```

Nous obtenons une suite de réponses générées par le `print` de notre `requests.put` :

```
<Response [201]>
```

Dans Futon, nous trouvons bien notre base de données, qui contient une suite d'articles. La figure 7-5 représente un morceau d'article tel que visualisé dans Futon.

Figure 7-5
Article visualisé dans Futon

Enfin, si nous accédons à notre base de données via l'interface REST, nous obtenons un état de la base à travers un retour structuré en JSON. Il nous suffit de saisir le code suivant :

```
curl http://127.0.0.1:5984/actualitte
```

qui retourne :

```
{"db_name":"actualitte","doc_count":30,"doc_del_count":0,"update_seq":30,"purge_
seq":0,"compact_running":false,"disk_size":122984,"data_size":90615,"instance_start_
time":"1347615883371079","disk_format_version":6,"committed_update_seq":30}
```

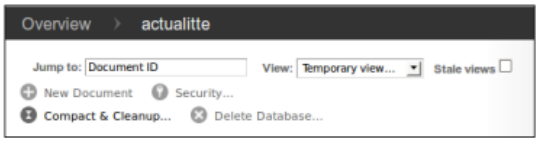
Notre base contient trente documents.

Fonctionnalités du serveur

Création et utilisation des vues

Maintenant que nous avons alimenté une base avec quelques documents, voyons comment utiliser les vues pour effectuer des recherches, ce que nous allons faire directement dans Futon. Pour cela, nous revenons à l'affichage de la base de données (http://127.0.0.1:5984/_utils/database.html?actualitte sur notre machine locale) et dans la liste déroulante View, nous sélectionnons Temporary view (figure 7-6).

Figure 7-6
Création d'une vue temporaire



Nous devons utiliser JavaScript, ou éventuellement CoffeeScript, un «surlangage» qui génère du JavaScript et dont la syntaxe est influencée par les langages de script comme Python et surtout Ruby. Une vue implémente les deux fonctions de MapReduce : la fonction `Map` est appliquée à chaque document, et la fonction `Reduce`, optionnelle, peut être utilisée pour retourner une agrégation.

Notre première vue sera vraiment très simple. Notre fonction `Map` sera la suivante :

```
function(doc) {
  if (doc.published) {
    emit(doc.published, doc);
  }
}
```

Nous allons simplement retourner nos documents organisés selon une nouvelle clé, qui correspond à la date de publication (`doc.published`).

Test d'existence de la clé

Dans notre fonction, nous testons l'existence de la clé `published` avant de retourner le document. C'est un élément important des vues dans CouchDB, car si la clé n'est pas présente dans tous les documents, les erreurs provoquées par le résultat de la fonction risquent de forcer CouchDB à désactiver l'indexation du résultat. En testant l'existence de la clé, nous nous assurons de ne retourner que les documents qui ont la structure qui nous intéresse. C'est une manière élégante de prendre en compte des documents de structures différentes dans la même base de données.

La seule fonction que nous appelons dans ce code très simple est `emit()`. Elle équivaut au `return` d'une fonction (à la différence qu'`emit()` peut être appelée plusieurs fois si vous voulez retourner plusieurs documents à partir d'une seule itération de la «boucle» `map`) et permet l'affichage d'un document en retour de la fonction. Les deux paramètres d'`emit()` correspondent à la structure d'un document : d'abord la clé, puis le document lui-même. Le rôle de la partie `map` d'une vue consiste à récupérer la liste des documents retournée par les exécutions d'`emit()` et à restituer cette liste ordonnée par la clé, donc la valeur du premier paramètre d'`emit()`.

Comme cette vue est temporaire, son résultat est dynamique, c'est-à-dire que l'application de la fonction `map` est réalisée à l'appel de la vue pour tous les documents de la base de données. Elle sera donc assez lente à exécuter. Les vraies vues, non temporaires, sont des vues matérialisées: lors de leur premier appel, elles stockent le résultat de la fonction `map` dans un B-Tree, de façon permanente sur le disque, et les modifications de données y sont conservées en temps réel lors des ajouts et mises à jour des documents. En d'autres termes, les vues ressemblent très fortement aux vues matérialisées des SGBDR. Pour créer ces vues durables, nous allons utiliser des documents de conception *(design documents)*.

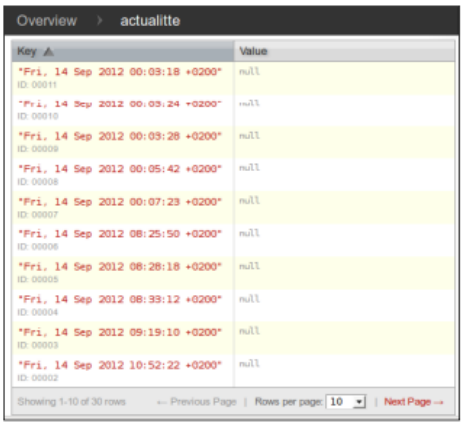
La vue peut très simplement faire office d'index secondaire. Voyons une transformation de notre vue temporaire :

```
function(doc) {
  if (doc.published) {
    emit(doc.published, null);
```

```
    }
  }
```

La seule différence est que nous avons retiré le second argument de la fonction `emit()`, nous renvoyons un `null`. En exécutant la fonction, nous obtenons le résultat illustré sur la figure 7-7.

Figure 7-7
Résultat de la vue



La partie Value du résultat ne contient rien, mais dans la partie Key, nous avons la clé définie dans notre fonction ainsi que l'ID du document d'origine. L'ID est automatiquement ajouté dans le résultat d'une vue afin de faire référence à ce document, ce qui correspond à l'usage d'un index secondaire.

Design documents

Les design documents constituent l'un des éléments les plus intéressants de CouchDB. Ils représentent une forme élégante de traitement des données au niveau du serveur, en offrant des fonctionnalités semblables à ce qu'on trouve dans un moteur SQL : vues, procédures stockées, déclencheurs et contraintes.

Nous modifions la vue temporaire que nous avons créée dans Futon, en inscrivant ceci dans la fonction `map` :

```
function(doc) {
  if (doc.tags.length > 0) {
    for(var i in doc.tags) {
      emit([doc.tags[i].term, doc.author_detail.name], 1);
    }
  }
}
```

Vous voyez dans cette modification de la fonction que nous pouvons manipuler des listes du document. Nous bouclons ici dans la collection de tags (les mots-clés) de chaque article et nous effectuons un `emit()` pour chaque terme trouvé. Le résultat de la fonction peut donc potentiellement retourner plus de paires clé-valeur que le nombre de documents originels. Nous voyons également que la clé que nous définissons dans la fonction `emit()` peut elle-même être composée d'une liste. Ensuite, nous ajoutons ceci dans la partie `reduce` :

```
function(keys, values, rereduce) {
  return sum(values)
}
```

Puis nous utilisons le bouton Save As… pour sauvegarder notre vue dans un design document (figure 7-8).

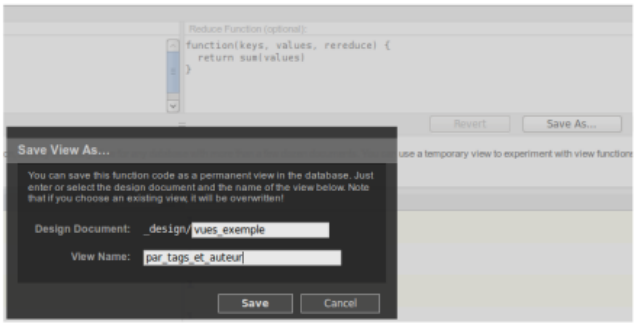
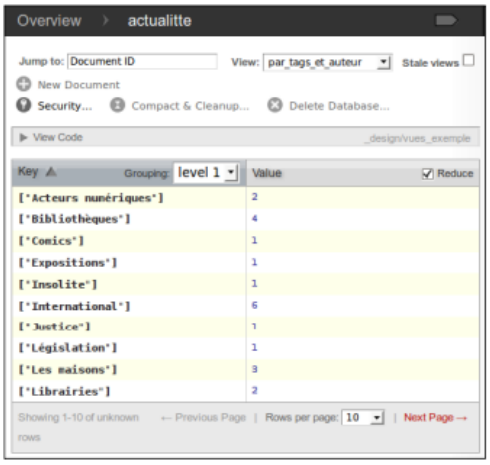


Figure 7-8
Sauvegarde de la vue

Une boîte de dialogue s'ouvre pour nous permettre de choisir le nom de notre design document et le nom de la vue qui y sera créée. Un design document peut contenir plusieurs vues, et il sera toujours préfixé par `_design/`, considérez cela comme une forme d'espace de noms. Le document créé sera un vrai document dans le sens de CouchDB et il contiendra du JSON. L'avantage de ceci est qu'il sera possible de le traiter comme les autres documents, dans toutes les opérations comme les mises à jour ou la réplication. Pour notre exemple, nous nommons le document `_design/vues_exemple` et la vue qui s'y trouve `par_tags_et_auteur`.

Nous pouvons maintenant utiliser la vue dans Futon. Nous revenons au niveau de la base de données `actualitte` et nous choisissons dans le menu déroulant View la vue `vues_exemple/par_tags_et_auteur`. Nous cliquons sur la case à cocher Reduce. Une liste déroulante nous permet alors de choisir le regroupement, nous sélectionnons level 1. Le résultat est illustré sur la figure 7-9.

Figure 7-9
Appel de la vue dans Futon



La fonction `map` nous a retourné une liste de clés basées sur un tag, puis le nom de l'auteur de l'article. En valeur, nous retournons simplement 1 afin de pouvoir en faire la somme dans la fonction `Reduce`. Lorsque nous appelons la vue, nous pouvons choisir ou non d'appliquer la fonction `Reduce`, et à quel niveau de regroupement. Nous avons effectué un agrégat de nos données, comme si nous avions écrit une requête SQL avec une clause `GROUP BY`.

Voyons enfin comment appeler cette vue depuis l'interface REST. L'appel de la vue se fait à l'adresse `base_de données/_design/nom_du_design_document/_view/nom_de_la_vue`.

```
curl
http://192.168.0.14:5984/actualitte/_design/vues_exemple/_view/par_tags_et_auteur
```

qui retourne ceci :

```
{«rows»:[
{«key»:null,»value»:30}
]}
```

Il s'agit donc de l'application de la fonction `Reduce` avec un niveau de regroupement total, ce qui équivaut à :

```
curl http://192.168.0.14:5984/actualitte/_design/vues_exemple/_view/
par_tags_et_auteur?group_level=0
```

Descendons ensuite dans les niveaux :

```
curl http://192.168.0.14:5984/actualitte/_design/vues_exemple/_view/
par_tags_et_auteur?group_level=1
```

ce qui retourne :

```
{«rows»:[
{«key»:[«Acteurs numériques»],»value»:2},
{"key":["Bibliothèques"],"value":4},
{"key":["Comics"],"value":1},
{"key":["Expositions"],"value":1},
{"key":["Insolite"],"value":1},
{"key":["International"],"value":6},
{"key":["Justice"],"value":1},
{"key":["Législation"],"value":1},
{"key":["Les maisons"],"value":3},
{"key":["Librairies"],"value":2},
{"key":["Récompenses"],"value":1},
{"key":["Scolarité France"],"value":1},
{"key":["Société"],"value":1},
{"key":["Univers BD"],"value":1},
{«key»:[«Usages»],»value»:4}
]}
```

Enfin, compliquons un peu notre dernier exemple :

```
curl --globoff 'http://192.168.0.14:5984/actualitte/_design/vues_exemple/_view/par_
tags_et_auteur?startkey=["I"]&endkey=["J"]&limit=3&group_level=2'
```

ce qui retourne :

```
{«rows»:[
{«key»:[«Insolite»,»Julien Helmlinger»],»value»:1},
{«key»:[«International»,»Ania Vercasson»],»value»:1},
{«key»:[«International»,»Clément Solym»],»value»:2}
]}
```

Nous avons cette fois demandé le dernier niveau, regroupé par les deux éléments de la clé, mais nous avons également effectué un filtre de rangée de valeurs. Nous avons exigé de retourner uniquement les clés qui se situent entre « I » et « J ». Cette rangée équivaut aux clés qui sont >= I et <= J. Cela nous permet de retourner uniquement les tags qui commencent par un I, puisqu'il n'y a aucun tag composé uniquement de la lettre J. Tout ce qui commence par J sera >J. De plus, nous limitons le résultat à trois à l'aide de l'instruction `limit`. Nous pouvons donc appliquer quelques filtres à nos vues.

Programmation client

Plusieurs bibliothèques Python permettent d'accéder à CouchDB, dont la liste est disponible à l'adresse suivante : http://wiki.apache.org/couchdb/Getting_started_with_Python.

Nous allons ici utiliser la bibliothèque `couchdbkit` (http://couchdbkit.org/). Elle est de bonne qualité et développée par un français, Benoît Chesneau. Nous l'installons avec `pip`, dans notre environnement virtuel `nosql`.

```
~/python_env/nosql/bin/pip install couchdbkit
```

La documentation de l'API est disponible ici : http://couchdbkit.org/docs/api/.

Voyons d'abord un exemple très simple d'utilisation. Nous allons insérer notre article de blog dans la base `passerelles` :

```
#!/usr/bin/python
# coding=utf-8

#import datetime
from couchdbkit import *

class Article(Document):
    auteur = DictProperty()
    titre = StringProperty()

db = Server() .get_or_create_db("passerelles")

Article.set_db(db)

art = Article(
    auteur = {'prénom': 'Annie',
        'nom': 'Brizard',
        'e-mail': 'annie.brizard@cocomail.com'},
    titre = 'pourquoi les éléphants ont-ils de grandes oreilles?'
)

art.save()
```

`couchdbkit` permet l'utilisation de classes Python pour représenter un document. La classe `Article` que nous créons hérite de la classe `couchdbkit.schema.Document`. Les types de données compatibles avec CouchDB et qui peuvent être déclarés comme propriétés de la classe sont dans le module `couchdbkit.schema`. Les plus importants sont repris dans le tableau 7-1.

Tableau 7-1. Types de données Couchbdkit

Types	Description
`StringProperty`	Une chaîne Ansi ou Unicode.
`IntegerProperty`, `LongProperty`	Un entier.
`FloatProperty`, `Number`, `DecimalProperty`	Un nombre réel (correspond aux types `float` et `decimal` en Python).
`BooleanProperty`	Un booléen.
`DateTimeProperty`, `DateProperty`, `TimeProperty`	Date complète, date seule et temps seul.
`DictProperty`	Un dictionnaire, corrrespondant à un dictionnaire Python et à un sous-document JSON.
`ListProperty`, `StringListProperty`	Une liste. La `StringListProperty` indique que la liste est composée uniquement de chaînes Unicode.

Nous avons ici utilisé un dictionnaire pour le sous-document qui concerne les détails de l'auteur, et une chaîne pour le titre. Nous créerons un dictionnaire anonyme au moment où nous attribuerons les valeurs de propriétés. Nous utilisons ensuite l'objet `Server` pour ouvrir une connexion. Nous utilisons un constructeur par défaut car nous nous connectons à la base locale. Nous aurions pu indiquer une URI pour une connexion distante et optionnellement une taille de batch, c'est-à-dire le nombre de documents à retourner en une fois. La signature partielle du constructeur est la suivante :

```
def __init__(self, uri='http://127.0.0.1:5984', uuid_batch_count=1000)
```

Nous utilisons la méthode `get_or_create_db()` de l'objet `Server` en lui passant le nom de la base de données. Cette méthode est pratique et dans l'esprit de la simplicité des bases NoSQL. Si vous souhaitez être plus rigoureux, les méthodes `get_db()` et `create_db()` existent également. Cette méthode nous retourne une instance de l'objet `Database` que nous attribuons à notre classe `Article` par l'intermédiaire de la méthode de classe `set_db()` qui la stocke dans un attribut de classe (l'équivalent d'une variable statique).

Nous créons ensuite une instance de notre classe `Article` et nous en définissons l'état dans le constructeur, puis nous utilisons la méthode `save()` de l'instance du document pour l'enregistrer dans CouchDB.

Nous appelons notre fichier `insertion.py` et nous l'exécutons dans notre environnement.

```
~/python_env/nosql/bin/python ~/insertion.py
```

Voilà pour ce qui concerne notre exploration de CouchDB. Étudions maintenant son successeur. Beaucoup de choses que nous venons de voir nous seront utiles pour comprendre comment fonctionne Couchbase Server.

Présentation de Couchbase Server

CouchDB n'est pas nativement distribué. Son architecture est plus adaptée à de petits développements qu'à des applications critiques. En revanche, sa richesse fonctionnelle est très intéressante.

Parallèlement, Membase, un moteur NoSQL orienté paires clé-valeur ditribué et en mémoire, était une initiative pour créer un système scalable et très rapide à partir de l'idée du serveur de cache, de type memcached. Couchbase Server est la réunion de ces deux moteurs dans un nouveau produit.

Caractéristiques

Couchbase Server est architecturé sur le modèle d'Amazon Dynamo : un cluster élastique et décentralisé. Il est simple d'ajouter ou de retirer des nœuds, et ensuite de lancer une opération de rebalance, qui va réorganiser les données sur le cluster automatiquement. Couchbase utilise le terme de bucket pour définir une base de données qui va contenir des paires clé-valeur. À la base, purement des paires clé-valeur à la valeur opaque. La jonction avec CouchDB a fait évoluer Couchbase vers un moteur orienté documents, qui reconnaît maintenant la valeur comme un document JSON.

Il existe deux types de buckets : memcached ou couchbase. Le bucket memcached offre un stockage uniquement en RAM, et aucune persistance sur le disque : vous pouvez donc l'utiliser pour bâtir un système de cache distribué avec Couchbase. Le bucket couchbase permet bien sûr la persistance sur le disque. Lorsque vous créez un bucket, vous définissez quel est sa taille en mémoire par nœud. Cela vous permet de réserver de la RAM par bucket pour dimensionner votre système. La mémoire totale du bucket sera donc la valeur par nœud multipliée par le nombre de nœuds. Vous indiquez également un coefficient de réplication, pour indiquer combien de réplicas vous voulez maintenir sur votre cluster. Vous ne pouvez évidemment pas indiquer un nombre supérieur à votre nombre de nœuds.

En interne, le bucket est découpé en vbuckets, selon le principe des vnodes dont nous avons parlé au chapitre 3. Rebalancer le cluster, cela veut dire échanger un certain nombre de vnodes entre nœuds.

L'accès aux données se fait normalement par la clé : on est ici dans un moteur clé-valeur. Lors d'un `get`, la clé est passée par une fonction de hachage qui identifie dans quel vbucket cette clé doit se trouver, et va donc diriger la demande vers un nœud. Les bibliothèques Couchbase peuvent être de type « *smart client* » et récupérer à la connexion la topologie du cluster, pour effectuer des get directement sur le bon nœud.

Fonctionnalités CouchDB

Couchbase intègre un certain nombre de fonctionnalités de CouchDB que nous venons de voir, notamment la création de documents de design et des vues matérialisées qui y sont contenues. Vous pouvez donc indexer vos données. Tout ceci semble très alléchant : un moteur distribué en mémoire, donc très rapide, avec toute la richesse fonctionnelle de CouchDB. Il y a quand même un ou deux bémols, qui ne sont pas forcément des défauts rédhibitoires mais plutôt des inconvénients de jeunesse, provenant de l'intégration des deux produits, et qui sont corrigés petit à petit au fil des versions.

Premièrement, lorsque vous accédez à un document par la clé, vous utilisez le protocole memcached binaire (https://code.google.com/p/memcached/wiki/BinaryProtocolRevamped) qui est très rapide. En revanche, lorsque vous accédez à une vue d'un document de design, vous appelez le moteur CouchDB intégré qui va, lui, demander un accès REST, donc beaucoup plus lent. En fin de compte, l'intégration des deux moteurs s'est faite sans redévelopper CouchDB dans le langage de Couchbase. Il y a donc deux moteurs qui s'exécutent sur chaque nœud : le moteur Couchbase développé en C++, et le moteur CouchDB en Erlang. D'ailleurs, sur les premières versions de Couchbase, la machine virtuelle Erlang (`beam`) avait une constante et intense activité CPU, même lorsque les nœuds n'étaient pas sollicités.

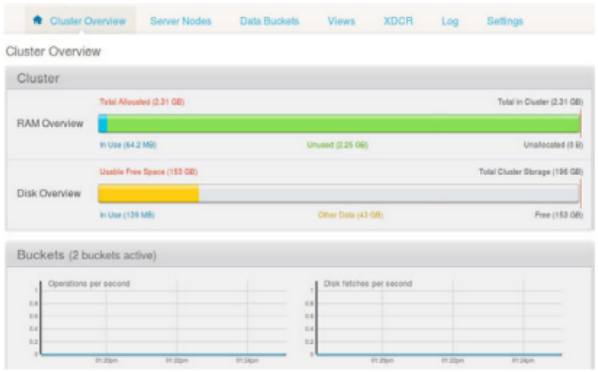
Comme vous accédez au moteur CouchDB, et que celui-ci ne gère aucun cache en mémoire, vos appels aux vues ne profitent donc pas du stockage en RAM.

Ces quelques exemples vous montrent une chose : si vous voulez utiliser Couchbase Server, intéressez-vous à l'avancement de l'intégration entre Membase et CouchDB dans la version actuelle, et utilisez Couchbase en connaissance de cause. Toutes les fonctionnalités n'offrent pas les mêmes performances. C'est de toute façon une règle à respecter en général : avant de choisir un moteur de bases de données pour votre production, prenez le temps de comprendre son architecture.

Interface d'administration

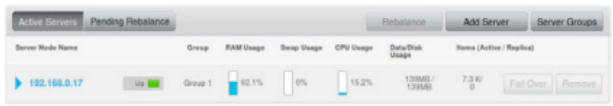
Un nœud Couchbase répond sur plusieurs ports selon le protocole utilisé. Le port 8091 est utilisé pour un accès à la console web, qui est un outil graphique d'administration, dont la page d'accueil est présentée sur la figure 7-10.

Figure 7-10
Couchbase Web Console



Vous pouvez y gérer votre cluster entier, en vous connectant à n'importe quel nœud, puisqu'il s'agit d'un système décentralisé. L'onglet Server Nodes (figure 7-11) vous permet de gérer vos nœuds, pour enlever, ajouter et rebalancer.

Figure 7-11
Gestion des nœuds



L'onglet Data Buckets vous permet de gérer les buckets et de visualiser les données (figure 7-12) et les vues (figure 7-13).

Figure 7-12
Visualisation des buckets

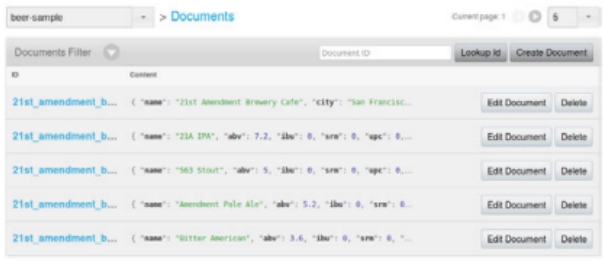
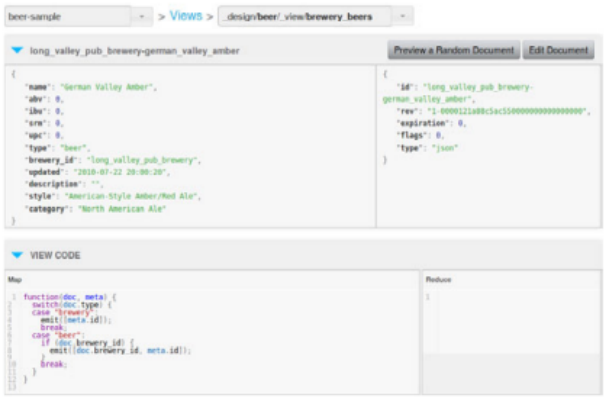


Figure 7-13
Gestion des vues



La console affiche également des compteurs et des statistiques par bucket fort utiles (figure 7-14).

Figure 7-14
Statistiques

Accès à Couchbase Server

Rien de surprenant ce ce côté : Couchbase développe des bibliothèques d'accès pour les langages les plus importants, en créant des smart clients qui récupèrent la topologie du réseau en se connectant à l'un des nœuds, ce qui leur permet ensuite d'aller directement au bon nœud pour un get sur la clé, et qui effectue un appel REST sur les nœuds en cas de requête sur des vues.

Le pilote Python nécessite la bibliothèque cliente Couchbase installée. Couchbase publie un dépôt Debian et Ubuntu. Nous récupérons la clé de signature du dépôt.

```
mkdir couchbase
cd couchbase/
wget http://packages.couchbase.com/ubuntu/couchbase.key
sudo apt-key add couchbase.key
```

Nous ajoutons l'adresse du dépôt dans un fichier source apt.

```
sudo vim /etc/apt/sources.list.d/couchbase.list
>deb http://packages.couchbase.com/ubuntu trusty trusty/main
```

Et nous installons la bibliothèque.

```
sudo apt-get update
sudo apt-get install libcouchbase2-core libcouchbase-dev
```

Enfin, nous installons le client Couchbase. Pour nous assurer d'avoir la dernière version, nous l'obtenons directement depuis son dépôt GitHub.

```
pip install git+git://github.com/couchbase/couchbase-python-client
```

Voici un exemple simple de code pour accéder à Couchbase. Nous indiquons à la connexion l'adresse de notre serveur Couchbase et le bucket que nous voulons utiliser.

```
from couchbase import Couchbase
cb = couchbase.Couchbase()
client = cb.connect("beer-sample", "192.168.0.17")

res = c.get("21st_amendment_brewery_cafe-north_star_red", quiet=True)
print res
```

L'option `quiet` du get supprime les messages d'erreur si la clé n'est pas trouvée.

N1QL

Un dernier mot sur Couchbase. Il s'agit d'un moteur ambitieux dont le développement est relativement rapide. Même s'il a encore quelques défauts de jeunesse, l'équipe de développement travaille ferme pour les corriger. Il s'agit déjà d'un moteur NoSQL important qu'il est intéressant de suivre. L'une des innovations récentes est l'introduction d'un langage de requête déclaratif à la syntaxe proche du SQL, nommé N1QL (à prononcer comme « nickel »).

On s'attend donc bien sûr à un `SELECT`, etc. Mais nous manipulons du JSON, n'est-ce pas ? Comment exprimer une structure hiérarchique ? Couchbase a inventé des solutions ingénieuses pour cela.

Voici deux exemples tirés de la documentation (http://docs.couchbase.com/prebuilt/n1ql/n1ql-dp4/N1QLRef-DP4.pdf)

```
SELECT name
FROM contacts
WHERE ANY child IN children
SATISFIES child.age > 14 END
```

Ici, on cherche à l'intérieur d'un JSON qui comporte des sous-éléments nommés `children`, avec un élément `age`. On souhaite donc tous les contacts qui ont au moins un enfant de plus de dix ans. Si on voulait les contacts dont tous les enfants ont plus de dix ans, on utiliserait le mot-clé `EVERY` au lieu de `ANY`.

```
SELECT purchases.purchaseId, l.product
FROM purchases
UNNEST purchases.lineItems l
WHERE DATE_PART_STR(purchases.purchasedAt,"month") = 4
AND DATE_PART_STR(purchases.purchasedAt,"year") = 2014
AND EXISTS (SELECT product.productId
    FROM product USE KEYS l.product
    WHERE product.unitPrice > 500);
```

Ici, on utilise la commande `UNNEST` pour décomposer les sous-éléments dans une sorte de jointure. Ces sous-éléments sont aplatis dans le résultat qui retourne donc, tout à fait comme avec une jointure traditionnelle, une ligne par `lineItems` avec les informations de `purchases`. Vous voyez également que les sous-requêtes sont supportées: elles agissent sur les données désimbriquées. C'est pour cette raison que l'on doit ajouter l'instruction `USE KEYS` afin d'indiquer quelle est la clé de chaque élément.

N1QL est donc un langage adapté pour les données hiérarchiques, et il s'agit d'une innovation très intéressante dans le monde NoSQL.

8

MongoDB

MongoDB est l'un des systèmes de gestion de données NoSQL les plus populaires. Développé par la société new-yorkaise MongoDB Inc., il est disponible depuis 2009. À l'heure où nous écrivons ces lignes, la dernière version disponible est la 2.6.8.

Présentation

Avant de fonder 10gen (renommée depuis en MongoDB Inc.), Kevin Ryan et Dwight Merriman avaient déjà créé DoubleClick, une entreprise bien connue de publicité en ligne. Frustrés par l'impossibilité de monter en charge comme ils le souhaitaient, ils décidèrent de développer une plate-forme applicative distribuée conçue pour le cloud, composée d'un moteur applicatif et d'une base de données, qu'ils développèrent d'abord en JavaScript. Cette idée de plate-forme applicative (un *App Engine* tel que celui de Google) n'intéressait pas grand monde, mais la base de données éveillait la curiosité, si bien qu'après un an, ils décidèrent de libérer son code. Ce fut à ce moment qu'un net intérêt commença à se manifester. Cet engouement se traduisit par une activité importante de développement par la communauté et un investissement grandissant de la part de 10gen, notamment par le recrutement d'ingénieurs spécialisés en SGBD.

Caractéristiques

MongoDB est l'un des rares systèmes de gestion de données NoSQL codés avec un langage qui offre de grandes performances : le C++. Les autres moteurs NoSQL populaires sont souvent codés en Java ou dans des langages particuliers comme Erlang.

MongoDB Inc. le définit comme un SGBD orienté documents. le terme « documents » ne signifie pas des documents binaires (image ou fichier son, par exemple), mais une représentation structurée, compréhensible par MongoDB, d'une donnée complexe. Plus simplement, on stocke la

représentation des données comprise par MongoDB. Ce format est appelé BSON (*Binary Serialized dOcument Notation* ou *Binary JSON*). Pour l'utilisateur, la structure visible est du JSON (*JavaScript Object Notation*, un format de données textuel dérivé de la syntaxe d'expression des objets en JavaScript).

Structure des données

Un document BSON représente donc l'unité stockée par MongoDB, qui équivaut à une ligne dans une table relationnelle. Il est composé d'une hiérarchie de paires clé-valeur, sans contrainte de présence ou de quantité. Le code suivant illustre un exemple de document basé sur notre application web de test : un site d'information communautaire, nommé *Passerelles*, qui publie des articles sur différents sujets, rédigés par les membres du site et que les lecteurs peuvent évaluer et commenter :

```
{
«auteur»:
{
    «prénom»:»Annie»,
    «nom»:»Brizard»,
    «e-mail»:»annie.brizard@cocomail.com»
},
«titre»:»pourquoi les éléphants ont-ils de grandes oreilles?»,
«mots-clés»:
[
    «faune»,
    «afrique»,
    «questions intrigantes»
],
«catégories»:
[
    «sciences»,
    «éducation»,
    «nature»
],
«date_création»: «12/05/2012 23:30»,
«statut»:»publié»,
«nombre_de_lectures»:»54»,
«commentaires»:
[
    {
        «date»:»14/05/2012 10:12»,
        «auteur»:»fred92@jmenmail.fr»,
        «contenu»:»il n'y a pas que les éléphants qui ont des grandes oreilles !»
    }
]
}
```

Vous reconnaîtrez la syntaxe JSON. Ce document sera stocké par MongoDB dans un format plus compact, plus pratique pour le stockage et les traitements, le BSON. À travers les bibliothèques

d'accès à MongoDB et l'invite interactive, ce format interne vous est masqué : vous travaillez dans un JSON lisible. À l'heure actuelle, la taille maximale d'un document est de 16 Mo.

Les documents sont contenus dans des collections, qui correspondent plus ou moins aux tables des SGBDR. Même s'il n'y a aucune obligation pour les membres d'une collection de partager une structure identique, il est tout de même plus logique – et plus efficace, parce que cela permettra l'indexation – de s'assurer de cette cohérence. MongoDB n'a pas de concept de contraintes comme dans le modèle relationnel. Il n'y a pas de vérification de cohérence effectuée par le serveur, toute la responsabilité est endossée à ce niveau par le code client. MongoDB stocke les documents tels qu'il les reçoit. De plus, il est inutile de créer explicitement une collection avant d'y insérer des documents, car MongoDB en crée une dès que le code client la mentionne.

Il y a tout de même une contrainte nécessaire : chaque document est identifié par une clé unique dans la collection, tout à fait dans le même esprit que les clés primaires des SGBDR. Il est bien entendu nécessaire de pouvoir identifier sans ambiguïté un document afin d'effectuer des mises à jour ou des suppressions. La clé porte un nom fixe dans le document : `_id`. Sa valeur peut être fournie ou générée automatiquement. Cela s'apparente au concept de clés techniques dans les systèmes relationnels, à la différence près qu'il ne s'agit pas ici d'une incrémentation automatique, mais de la génération d'un `UUID` (un type nommé `ObjectId` dans MongoDB, d'une taille de 96 octets), c'est-à-dire une valeur créée par un algorithme qui garantit une très faible probabilité de collision (c'est ainsi qu'on appelle la génération d'un doublon), un peu comme le `GUID` *(Globally Unique IDentifier)* de Microsoft. Cette approche permet d'assurer un couplage faible et un fonctionnement correct dans le cas d'une base de données distribuée.

Les collections peuvent être regroupées dans des espaces de noms, et sont stockées dans des bases de données *(databases)*. Un espace de noms est simplement un préfixe ajouté aux collections (et séparé de celles-ci par un point), qui permet de les regrouper, un peu comme le concept de schémas de la norme SQL.

Correspondance avec le Schéma

D'ailleurs, la comparaison avec la norme SQL peut être poussée. En effet, cette dernière contient dans sa section ISO/IEC 9075-11:2008 (SQL/Schemata) une définition d'informations de catalogue dans le schéma `INFORMATION_SCHEMA`, notamment implémenté dans MySQL, MS SQL Server ou PostgreSQL. MongoDB offre un mécanisme similaire avec un espace de noms `system` présent dans chaque base de données et qui contient des collections de métadonnées.

Une base de données peut être considérée comme une collection de collections. Elle est stockée indépendamment sur le disque, et le concept est assez proche de ce qu'on appelle une base de données dans le monde relationnel, comme dans MySQL.

Comme pour les collections, ces bases de données n'ont pas besoin d'être préalablement créées, elles le seront dès qu'une commande les mentionnera. Il n'y a d'ailleurs pas de commande pour créer une base de données. Il vous suffit d'utiliser une base de données (`use mabase`). Cette approche est très libérale et demande de la rigueur au programmeur au lieu de lui imposer des garde-fous.

Stocker des objets larges

MongoDB peut également stocker des documents binaires ou des objets larges, soit directement dans sa structure BSON, soit à l'aide de GridFS, spécification destinée à optimiser la manipulation et la sérialisation de fichiers de taille importante. En interne, MongoDB découpe les fichiers en plus petits segments qui seront autant de documents contenus dans une collection nommée `chunks`. Les métadonnées des fichiers sont conservées dans une autre collection, nommée `files`. Vous pouvez ajouter des attributs aux documents de `files` pour qualifier vos fichiers. La collection `chunks` contiendra des documents qui auront un `files_id` identifiant le document de la collection `files`, un attribut `n` qui est le numéro du segment, et un attribut `data` qui correspond au segment, de type binaire. C'est une façon optimale de conserver des fichiers par rapport à un stockage sur un système de fichiers avec des URL. Un outil en ligne de commande, `mongofiles`, permet d'importer ou d'extraire des documents.

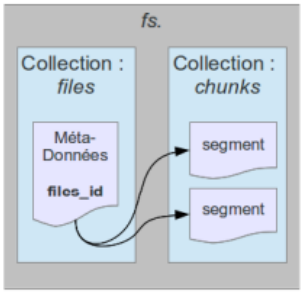


Figure 8-1
GridFS

Le traitement des données

Du point de vue transactionnel, MongoDB supporte l'atomicité seulement lors de la mise à jour d'un document. En d'autres termes, il y a bien un verrouillage à l'écriture sur un document, mais il ne s'étend pas par défaut sur une mise à jour de plusieurs documents.

En ce qui concerne la durabilité des transactions, elle est assurée par une journalisation de forme `WAL, write-ahead log`. Les instructions de création-modification-suppression de données sont en premier lieu écrites dans un journal. Si le moteur est arrêté proprement, les écritures sont inscrites sur le disque et le journal est supprimé. Donc si MongoDB démarre et constate l'existence d'un journal, c'est que le moteur n'a pas été arrêté correctement. Il exécute alors une phase de récupération qui rejoue le journal. La journalisation est activée par défaut dans les versions récentes et 64 bits de MongoDB, elle peut être désactivée au démarrage par l'option de la ligne de commande `--nojournal`. Vous perdez en sécurité et gagnez un peu en performances lors des écritures.

Mise en œuvre

Installation

Les exécutables MongoDB sont disponibles à l'adresse suivante : http://www.mongodb.org/downloads. L'installation sur Windows ou Mac OS X est très simple. Étant donné que nous disposons d'une distribution Ubuntu Server pour nos exemples, nous pouvons utiliser le paquet `mongodb` de la distribution officielle (MongoDB est disponible sur la plupart des dépôts officiels des distributions Linux).

Pour rester à jour avec les dernières versions, nous allons plutôt choisir le paquet réalisé par MongoDB Inc., disponible sur downloads-distro.mongodb.org. Les commandes pour ajouter le dépôt et installer MongoDB sont les suivantes :

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
sudo apt-get install python-software-properties
sudo apt-add-repository "deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart
dist 10gen"
sudo apt-get update
sudo apt-get install mongodb-org
```

Le dépôt porte toujours le nom de 10gen, mais depuis quelques versions, le paquet s'appelle `mongodb-org`. Auparavant, son nom était `mongodb-10gen`.

Le dépôt `ubuntu-upstart` installe une version du démon `mongodb` qui se lance grâce au système upstart d'Ubuntu, un système de lancement de services remplaçant le traditionnel démon `init` issu de System-V. Ainsi, le démarrage et l'arrêt de MongoDB s'exécuteront de la façon suivante :

```
sudo start mongodb
sudo stop mongodb
sudo restart mongodb
sudo status mongodb
```

Documentation pour Debian et Ubuntu

L'installation et le démarrage sur une distribution Debian ou Ubuntu non upstart sont décrits à l'adresse suivante : http://docs.mongodb.org/manual/installation/.

La configuration du démon s'effectuera ensuite dans le fichier `/etc/mongodb.conf`. Nous conservons ici les valeurs par défaut, qui suffisent pour notre découverte de MongoDB. Vous pourrez également modifier le script de démarrage du service : `/etc/init/mongodb.conf`, ou encore lancer des démons manuellement (voir plus loin la section sur la réplication page 197).

L'invite interactive

L'accès à une invite de commandes interactive pour manipuler les données de MongoDB est possible grâce au shell `mongo` (`/usr/bin/mongo` sur notre Ubuntu). Il s'agit d'une invite JavaScript (qui utilise l'interpréteur V8 depuis la version 2.4 de MongoDB, SpiderMonkey auparavant) grâce à

laquelle vous pouvez interagir avec MongoDB. Pour l'ouvrir, tapez simplement `mongo` dans votre terminal, l'exécutable devrait se trouver dans le chemin. Si vous voulez accéder au serveur local qui écoute sur le port par défaut 27017, cela suffira. Sinon, vous pouvez indiquer l'adresse sous la forme suivante : `<serveur>:<port>/<base de données>`.

Voyons ensuite quelques commandes :

```
show dbs;
use passerelles;
show collections;
db.articles.find();
it;
```

Nous affichons tout d'abord la liste des bases de données. Nous entrons ensuite dans la base `passerelles`, nous listons les collections de cette base et enfin nous affichons les dix premiers objets de la collection `articles`.

Existence préalable de la base

Même si la base et la collection ne sont pas créées, vous pouvez exécuter les commandes du code précédent. Utilisez `use passerelles`, par exemple, pour entrer dans le contexte de la base `passerelles`. Si vous demandez un `show dbs`, vous ne verrez pas la base `passerelles` dans la liste tant que vous n'aurez pas créé un objet à l'intérieur.

Par la commande `it`, nous affichons les dix objets suivants. Nous pouvons également utiliser une boucle `while` ou la méthode `forEach()` de l'objet curseur retourné par `find()` :

```
# première façon
var cur = db.articles.find();
while (cur.hasNext()) printjson(cur.next());
# deuxième façon
db.articles.find().forEach(printjson);
```

Comme il s'agit d'un interpréteur JavaScript interactif, nous pouvons mélanger du code ou créer des objets JavaScript pour les insérer dans nos collections, ils seront automatiquement transformés en BSON par le shell.

Obtention d'aide

Pour chaque classe ou méthode, vous pouvez demander la méthode `.help()` qui retournera la liste des méthodes disponibles ou une aide syntaxique.

Créons par exemple un index sur une valeur de notre structure JSON :

```
db.articles.ensureIndex({date_création=-1});
```

Ici, nous demandons la création d'un index à travers un document qui décrit les clés participant à l'index (ici, l'attribut `date_création`), l'index peut donc être composite. L'ordre est ensuite indiqué par 1 (ascendant) ou -1 (descendant). Nous préférons un ordre descendant car nous allons plutôt

requêter les articles en les triant par la date de publication, de la plus récente à la plus ancienne. Les index sont des structures en arbres équilibrés (B-trees) qui vont permettre des recherches dichotomiques, une structure traditionnelle des méthodes d'indexation. La méthode `ensureIndex()`, comme son nom l'indique, s'assure que l'index est bien présent. Elle est utile car elle ne crée pas un deuxième index si l'attribut est déjà indexé.

Les index peuvent être définis comme *sparse* : seuls seront indexés les documents qui contiennent les attributs spécifiés dans l'index.

Insérons notre document. Pour cela, le plus simple est d'enregistrer la commande d'insertion avec la structure du document dans un fichier `.js`, que nous évaluerons avec le shell `mongo`, soit en le passant à la ligne de commande en lançant le shell, soit dans une session de l'invite, par la commande `load()` :

```
load(«newdoc.js»);
```

Cette commande d'insertion créera simplement un objet JavaScript comprenant notre document :

```
var doc = {
«auteur»:
[…]
};
```

Code abrégé

Nous avons raccourci le code pour en améliorer la lisibilité. L'intégralité du code est disponible sur www.editions-eyrolles.com à la page consacrée au livre.

Utilisons ensuite la méthode `insert()`, ou mieux, la méthode `save()` :

```
db.articles.save(doc);
```

`save()` exécute un *upset*, c'est-à-dire qu'il met à jour ou insère selon que la valeur de `_id` (si elle est explicite) est trouvée dans la collection.

Nous avons vu quelques commandes simples de l'invite. Vous trouverez une description complète du shell sur le site de MongoDB Inc. à l'adresse : http://docs.mongodb.org/manual/administration/scripting/. La plupart du temps, vous ne ferez pas d'insertion à l'aide de l'invite, mais à partir d'un code client. Quittons ce shell en tapant `exit`, ou à l'aide de la combinaison de touches Ctrl + D, et voyons comment accéder à MongoDB depuis notre code Python.

Programmation client

MongoDB Inc. développe des bibliothèques pour de nombreux langages, appelés pilotes *(drivers)*. Celui pour Python se nomme PyMongo, il s'agit d'un pilote officiel. Avant de l'installer, nous nous assurons que le paquet `python-dev` est installé sur notre serveur, car il permettra de compiler les extensions en C du pilote et d'améliorer sensiblement ses performances :

```
sudo apt-get install python-dev
```

Nous installons ensuite le pilote avec `pip`, dans notre environnement virtuel `nosql` :

```
~/python_env/nosql/bin/pip install pymongo
```

Voyons un exemple simple d'utilisation, dans un fichier source Python, en créant d'abord un dictionnaire Python qui contient le document que nous allons stocker :

```
#!/usr/bin/python
# coding=utf-8

from datetime import datetime

article = {
«auteur»:
[…]
}

«»» ouvrons une connexion sur mongodb «»»
import sys
import pymongo
from pymongo import Connection
from pymongo.errors import ConnectionFailure

def main():
try:
    cn = Connection(host=»localhost», port=27017)
    print «connexion réussie»
except ConnectionFailure, e:
    sys.stderr.write(«Erreur de connexion à MongoDB:%s»% e)
    sys.exit(1)

db = cn[«passerelles»]
assert db.connection == cn
print «handle acquis avec succès»

db.articles.insert(article, safe=True)

if __name__ == “__main__”:
main()
```

Nous avons ici effectué notre écriture avec l'instruction `db.articles.insert(article, safe=True)`, ce qui signifie : dans la base de données `db` ouverte, dans la collection `articles`, insérer le dictionnaire `article`. L'objet sera automatiquement transformé en document JSON par PyMongo. Si la collection `articles` n'existe pas, elle sera créée.

Encodage dans Python

Nous avons indiqué à Python quel est le codage utilisé dans le script, sur la deuxième ligne :

```
# coding=utf-8
```

Sans cela, Python risque de ne pas pouvoir interpréter les signes diacritiques (les caractères accentués) dans le document.

Le paramètre `safe=True` indique d'effectuer l'insertion en mode synchrone. Le pilote PyMongo effectue l'insertion en mode asynchrone par défaut, ce qui est pour le moins une option dangereuse pour une application sérieuse : une erreur resterait silencieuse. Une autre option permet de s'assurer que l'écriture est bien effectuée sur un nombre défini de nœuds dans une configuration en *replica set* :

```
db.articles.insert(article, w=2)
```

Ici, nous indiquons que l'écriture doit avoir été réellement effectuée sur deux nœuds avant de redonner la main. Cette option implique bien sûr le mode synchrone, il est donc inutile de spécifier `safe=True`.

Write-concern

Le paramètre `w` peut être utilisé de différentes façons : en spécifiant une étiquette qui indique une règle prédéfinie ou en utilisant le mot-clé `majority` pour demander une validation par une majorité de nœuds. La majorité est un nombre de nœuds qui évolue automatiquement avec le nombre de réplicas.

Il nous suffit donc d'exécuter notre fichier Python dans notre environnement (ici, nous avons appelé le fichier `insertion.py`) :

```
~/python_env/nosql/bin/python ~/insertion.py
```

Extraire et chercher

Une requête est écrite en JSON et spécifie, à l'aide d'opérateurs, les valeurs recherchées dans le document. Cherchons ici les articles publiés dans la catégorie « Éducation » et qui comportent le mot-clé « afrique » :

```
query = {
«mots-clés»:»afrique»,
«catégories»:»éducation»
}
```

Comme vous le voyez, la requête est effectuée en créant un document de recherche, qui décrit les attributs qui doivent être présents pour que le document soit retourné.

Une recherche renvoie un curseur (un objet `Cursor`) qui permet l'itération sur les documents trouvés. La méthode qui retourne ce curseur s'appelle `find()` :

```
articles = db.articles.find(query)
for article in articles:
```

```
print article.get(«auteur»)
```

Cette requête peut indiquer un filtrage et un tri des clés des membres retournés :

```
articles = db.articles.find(
{«catégories»:»éducation»}).sort((«date_création», pymongo.DESCENDING))
```

Correspondance avec le langage SQL

Si on veut établir une correspondance avec le monde SQL, on peut dire que le filtrage correspond à l'indication des colonnes dans le `SELECT` (ou à une projection en algèbre relationnelle) et que le `sort()` correspond à un `ORDER BY`.

Vous pouvez également utiliser la clause `limit()`, pour ne retourner qu'un nombre défini de résultats :

```
articles = db.articles.find(
{«catégories»:»éducation»}).limit(10)
```

Quand vous avez la certitude que votre requête ne retourne qu'un seul document, vous pouvez utiliser la méthode `find_one()` qui vous économise le traitement du curseur :

```
article = db.articles.find_one({«mots-clés»:»afrique»,
«catégories»:»éducation»,
«date_création»: «12/05/2012 23:30»})
print article.get(«auteur»)
```

Les agrégations

L'un des principaux points faibles de MongoDB (et de beaucoup de moteurs NoSQL) était du côté de l'analytique. Souvent, les entreprises qui adoptent un moteur NoSQL pensent d'abord aux besoins opérationnels : maniupler des données unitaires en lecture et en écriture. Mais rapidement, lorsque la solution NoSQL est en production, des besoins analytiques se font sentir : calculer des statistiques et des agrégats sur les données. Cela veut dire avoir un moyen de faire des calculs sur tous les membres d'une collection, du côté serveur, sans avoir à récupérer ces données sur le client. Historiquement, MongoDB offrait la possibilité d'envoyer des fonctions MapReduce écrites en JavaScript sur le serveur. Malheureusement, l'interpréteur JavaScript côté serveur n'est pas très performant. Pour améliorer les choses, MongoDB a remplacé l'ancien moteur JavaScript SpiderMondey par V8 de Google. C'est plus rapide, mais pas suffisamment.

Pour cette raison, MongoDB a développé et intégré un moteur d'agrégation appelé simplement *aggregation* (http://docs.mongodb.org/manual/aggregation/). Il est basé sur la notion de pipeline, qui calcule des agrégats et les envoie éventuellement vers d'autres calculs d'agrégats avant de produire le résultat. La commande `aggregate()` prend en paramètre une définition d'étapes d'agrégations et exécute le tout sur le serveur. Voici un exemple simple tiré de la documentation :

```
db.zipcodes.aggregate( [
   { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
```

```
    { $match: { totalPop: { $gte: 10*1000*1000 } } }
] )
```

Vous y voyez deux opérations, d'abord un regroupement effectué sur le champ `state` des documents de la collection, avec extraction de la somme de la population par `state` dans un champ qui est nommé au passage `totalPop` (une forme d'alias). Ensuite, un filtre `$match` est appliqué pour ne conserver que les états dont la population est supérieure à dix millions. Le filtre `$match` est l'équivalent ici du `HAVING` en SQL.

Administration

L'interface web d'administration est disponible à l'adresse http://localhost:28017. Vous pouvez appeler cette page depuis une machine cliente à condition que la configuration de MongoDB le permette. Dans le fichier `/etc/mongodb.conf`, vous pouvez indiquer sur quelle interface le serveur va écouter avec l'option `bind_ip`. Pour autoriser un accès à partir de toutes les interfaces, enlevez ou commentez cette ligne de configuration. Vous devez bien entendu redémarrer le démon pour que le changement soit pris en compte.

Activation de l'interface REST

La plupart des fonctionnalités de l'interface d'administration web nécessitent que le démon `mongodb` soit démarré avec l'option `REST`. Pour cela, activez ou ajoutez la ligne suivante sur Ubuntu :

```
rest = true
```

dans `/etc/mongodb.conf`. Les tâches d'administration que requiert MongoDB sont du domaine de ce qu'on appelle dans les SGBDR un DBA de production, pas d'un DBA études. En d'autres termes, vous aurez à effectuer des sauvegardes, des restaurations, à sécuriser l'accès, mais vous devrez rarement créer des bases et des schémas de données.

Les opérations d'administration peuvent être exécutées grâce à l'invite interactive ou par l'intermédiaire d'un pilote. À l'intérieur d'une base de données, un espace de noms spécial, `$cmd`, permet de passer des commandes à travers sa méthode `FindOne()`, qui retourne son résultat sous forme de document. Beaucoup de ces commandes ont des fonctions clairement nommées qui les encapsulent. Par exemple, l'invite interactive dispose d'une fonction générique `db.runCommand()` et de fonctions spécifiques. Certaines de ces commandes sont réservées à l'administrateur et sont disponibles dans la base de données `admin` uniquement.

```
use admin;
db.runCommand("shutdown");
```

Il y a souvent des fonctions qui encapsulent ces appels :

```
use admin;
db.shutdownServer();
```

Vous trouvez également un client graphique en développement qui est prometteur : Robomongo. http://robomongo.org/.

Sécurité

Qu'en est-il de la sécurité dans MongoDB, dans le sens où on l'entend pour tout type de serveur: authentification et protection des données ? Pour l'authentification, la configuration recommandée par MongoDB Inc. est de n'en avoir aucune et d'exécuter le ou les nœuds MongoDB dans un environnement sûr *(trusted environment)*. C'est par la configuration réseau que vous vous assurez que les ports TCP permettant d'accéder à MongoDB ne sont accessibles que depuis des machines de confiance. Ces clients peuvent accéder et manipuler les données comme bon leur semble.

Utilité pour les besoins simples

Nous avons une fois de plus l'illustration du caractère permissif et décontracté de certaines bases de données NoSQL. Elles ne sont pas conçues pour stocker et protéger des données sensibles. Il faut les envisager comme des dépôts de données et non comme des systèmes de gestion de bases de données dans le sens où on l'entend traditionnellement. Cela illustre aussi l'intérêt de MongoDB en tant que conteneur de données pour les sites web, même modestes, un peu à la manière de MySQL : cela peut être aussi un outil léger pour des besoins simples.

Au-delà de cette liberté, MongoDB peut aussi gérer une sécurité d'accès par base de données, très basique à l'heure actuelle. Un utilisateur peut être créé, soit avec toutes les permissions, soit avec une restriction de lecture seule. La connexion devra être effectuée en mentionnant les paramètres d'authentification dans la chaîne de connexion :

```
mongodb://amelie@ViZiGoth4452@localhost/passerelles?journal=true
```

Ici, nous indiquons par la chaîne de connexion que l'utilisateur `amelie`, mot de passe `ViZiGoth4452`, se connecte au serveur local, dans la base de données `passerelles`. Nous ajoutons le paramètre `journal=true` : la journalisation sera activée pour cette session.

Montée en charge

MongoDB se veut un outil pratique répondant à des besoins concrets de développement. Il est à la fois simple à utiliser et conçu pour s'interfacer de façon naturelle avec les langages clients orientés objet.

Capped collections

Un exemple d'adaptation de MongoDB aux besoins courants des infrastructures informatiques est qu'il inclut un concept intéressant pour le stockage de journaux, qu'on pourrait traduire en français par « collection bouchonnée » *(capped collection)*. Il s'agit d'une collection à la taille fixe, prévue pour offrir d'excellentes performances et qui agit comme un pile qui maintient automatiquement un nombre fixe d'entrées, avec un comportement FIFO *(First-In First-Out)*, un concept similaire au RRD *(Round-Robin Database)* utilisé par des outils comme Nagios. Le nombre d'entrées reste constant et au fur et à mesure que de nouveaux documents sont insérés, les plus anciens sont automatiquement supprimés.

MongoDB est adapté à des besoins modestes de stockage de données, mais il est aussi un système de gestion de données conçu au départ pour être distribué. Cette distribution est possible

grâce à deux mécanismes: la réplication, en créant des ensembles de réplicas (replica sets) et la répartition (sharding).

Réplication

Un démon (processus) MongoDB est appelé un nœud. Dans les situations de production, il y a en général un nœud par serveur. Il peut être utile de monter une machine multinœud pour le développement et les tests.

Ces nœuds peuvent être liés dans un ensemble de réplicas, c'est-à-dire une topologie de réplication de type maître-esclave telle qu'on la trouve dans les SGBDR, à des fins de duplication de données, et pour assurer une continuité de service, car la réplication inclut également un basculement automatique du maître. À un moment donné, un nœud est choisi comme maître par MongoDB et reçoit donc toutes les écritures, qu'il distribue aux esclaves. En cas de défaillance du nœud maître, MongoDB bascule automatiquement et promeut un autre nœud maître. Les applications clientes sont averties de ce fait et redirigent toujours les écritures vers le bon maître dans un jeu de réplicas. Conceptuellement, cette réplication est effectuée en mode asynchrone ou synchrone selon la demande du client. Une opération de modification de données peut indiquer le nombre de réplicas sur lesquels l'écriture doit avoir été effectuée avec succès avant de rendre la main, comme nous l'avons vu dans les exemples de code précédents.

Le jeu de réplicas est constitué d'un nombre impair de processus, ce qui permet d'avoir toujours deux machines qui vont former quorum pour élire un nouveau maître. Si vous disposez d'un nombre pair de nœuds, vous pouvez ajouter un arbitre, qui est un démon `mongod` léger dont le but est uniquement de former quorum. Inutile de dédier un serveur pour l'arbitre, il peut très bien tourner sur une machine déjà occupée à autre chose.

Comme beaucoup de choses en MongoDB, la réplication est aisée à mettre en place. Il suffit d'abord de démarrer vos démons avec l'option `--replSet <NomDeRéplica>`, où `<NomDeRéplica>` est un nom conventionnel que vous choisirez identique pour un jeu de réplicas. Nous créons un environnement de tests et un jeu de réplicas sur la même machine:

```
sudo mkdir /var/lib/mongodb_datareplica1 /var/lib/mongodb_datareplica2 /var/lib/
mongodb_datareplica3
sudo mongod --dbpath /var/lib/mongodb_datareplica1 --port 15000 --replSet ReplicaSet1
> /tmp/mongo_replica1.log &
sudo mongod --dbpath /var/lib/mongodb_datareplica2 --port 15001 --replSet ReplicaSet1
> /tmp/mongo_replica2.log &
sudo mongod --dbpath /var/lib/mongodb_datareplica3 --port 15002 --replSet ReplicaSet1
> /tmp/mongo_replica3.log &
```

Sur des machines différentes, en situation de production, les options `--dbpath` et `--port` ne seront pas nécessaires.

Connectons-nous avec l'invite de commandes sur le premier serveur:

```
mongo --host localhost:15000
```

Nous créons un document de configuration, et nous initialisons la réplication:

```
config = {_id: 'ReplicaSet1', members: [
                {_id: 0, host: 'localhost:15000'},
                {_id: 1, host: 'localhost:15001'},
                {_id: 2, host: 'localhost:15002'}]
        };
rs.initiate (config);
```

ce qui retourne :

```
{
"info": "Config now saved locally. Should come online in about a minute.",
«ok»: 1
}
```

Changement de configuration en production

En situation de production, vous pouvez exécuter un `rs.initiate()` préliminaire sans configuration et ajouter vos serveurs un à un avec la fonction `rs.add(«serveur»)`, qui encapsule les opérations de changement de configuration. Mais ici, comme nous voulons que tous nos nœuds soient sur la même machine, nous devons modifier la configuration en une fois, car MongoDB nous force à la faire si nous voulons utiliser `localhost` comme nom de machine.

Ensuite, nous pouvons afficher la configuration et le statut de la réplication grâce aux commandes suivantes :

```
rs.conf()
rs.status()
```

Modification de l'invite

Notons que l'invite de commandes change et affiche `ReplicaSet1:PRIMARY` pour indiquer la connexion non plus à un serveur spécifique, mais au nœud primaire du jeu de réplicas.

Notre nœud primaire est celui qui écoute sur le port 15000. Quittons `mongo` et arrêtons ce nœud pour voir si le basculement fonctionne :

```
sudo pkill -f datareplica1
```

Entrons ensuite dans le deuxième serveur `mongo` et vérifions :

```
mongo --host localhost:15001
rs.isMaster()
```

ce qui donne :

```
{
«setName»: «ReplicaSet1»,
"ismaster": true,
"secondary": false,
```

```
"hosts": [
    «localhost:15001»,
    «localhost:15002»,
    «localhost:15000»
],
«maxBsonObjectSize»: 16777216,
«ok»: 1
}
```

Il y a bien eu basculement, ce que nous voulions démontrer. Si ce nœud n'avait pas été choisi comme maître, vous auriez vu de toute façon un `SECONDARY>` en invite de commandes. Le troisième nœud aurait été choisi, ce que vous auriez pu vérifier avec un `rs.status()`.

Répartition (sharding)

La montée en charge horizontale est réalisée par répartition (sharding). MongoDB peut aussi bien s'exécuter en tant que moteur individuel sur une seule machine ou être converti en un cluster de données éclatées géré automatiquement et comportant un nombre de nœuds pouvant théoriquement aller jusqu'à mille. Le sharding de MongoDB impose quelques limitations légères à ses fonctionnalités, notamment une immutabilité de la clé (primaire) du document. Le choix de la répartition est fait par collection. C'est une collection, et non pas une base de données, qu'on répartit. Les documents de cette collection seront distribués aussi équitablement que possible entre les différents nœuds, afin d'équilibrer les traitements. À chaque opération, il faut bien entendu identifier le nœud qui contient ou doit contenir les données. Ce rôle est assuré par un processus nommé `mongos`, qui reçoit les requêtes et les reroute, et qui agit donc comme un répartiteur de charge. Pour assurer une continuité de service, cette répartition peut être accompagnée de réplication : chaque nœud peut avoir son propre groupe de réplicas avec basculement automatique.

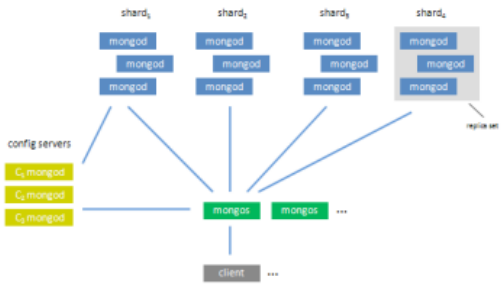


Figure 8-2
Le modèle de répartition de MongoDB (Source : documentation officielle, http://docs.mongodb.org/manual/sharding/)

La configuration de la répartition est maintenue par un ou plusieurs serveurs de configuration. MongoDB Inc. recommande d'en avoir trois, car le protocole de synchronisation entre les serveurs de configuration est optimisé pour trois serveurs.

Installer la répartition consiste à :

1. Démarrer des démons `mongod` avec l'option `--shardsvr`.
2. Démarrer un ou plusieurs démons `mongod` avec l'option `--configsvr` pour en faire des serveurs de configuration.
3. Démarrer un ou plusieurs serveurs `mongos` : `mongos --configdb <serveur>:<port>`. L'adresse des serveurs de configuration est passée en paramètre au démarrage de `mongos`.
4. Se connecter à un serveur `mongos` pour configurer les shards.

Voici un exemple en code de la configuration sur une seule machine qui simulera un environnement de répartition. Premièrement, nous lançons les différents nœuds.

```
sudo mkdir /var/lib/mongodb_data1 /var/lib/mongodb_data2
sudo chown -R mongodb.mongodb /var/lib/mongodb_data1
sudo chown -R mongodb.mongodb /var/lib/mongodb_data2
sudo mongod --shardsvr --dbpath /var/lib/mongodb_data1 --port 10000 > /tmp/shard1.log
&
sudo mongod --shardsvr --dbpath /var/lib/mongodb_data2 --port 10001 > /tmp/shard2.log
&
```

Vérifions que tout s'est bien passé :

```
cat /tmp/shard1.log
cat /tmp/shard2.log
```

Puis nous lançons un serveur de configuration :

```
sudo mkdir /var/lib/mongodb_config
sudo chown -R mongodb.mongodb /var/lib/mongodb_config
sudo mongod --configsvr --dbpath /var/lib/mongodb_config --port 20000 > /tmp/mongo_
config.log &
cat /tmp/mongo_config.log
```

Et enfin nous créons le démon `mongos` :

```
sudo mongos --configdb localhost:20000 > /tmp/mongos.log &
cat /tmp/mongos.log
```

En lisant les messages envoyés dans `/tmp/mongos.log` au démarrage, nous voyons que c'est le démon `mongos` qui écoute sur le port 27017, le port par défaut du serveur MongoDB. Un client accédera donc au serveur `mongos` qui s'occupera de rediriger les commandes sur le bon shard.

Bien entendu, ce n'est pas terminé, nous n'avons pas indiqué que les deux démons `mongod` que nous avons démarrés sont des nœuds. Pour cela, nous devons nous connecter sur le serveur `mongos` en invite de commandes en tapant `mongo`, qui montrera une invite `mongos>` puis, dans l'invite :

```
use admin
db.runCommand( { addshard: "localhost:10000" } )
db.runCommand( { addshard: "localhost:10001" } )
```

Nous activons ensuite la répartition pour la base de données `passerelles` :

```
db.runCommand( { enablesharding: "passerelles" } )
```

Nous devons ensuite choisir une clé de répartition pour nos collections, c'est-à-dire une représentation des éléments d'un document qui détermine la segmentation. Il s'agit simplement d'une liste d'attributs (un ou plusieurs), par exemple :

```
{ "date_création": 1; "catégories": 1 }
```

Ici, nous décidons de répartir par date de création et par catégorie d'articles. Il est important de choisir une bonne clé dès le début, car les changements postérieurs seront probablement douloureux. Nous choisissons une clé suffisamment fine pour donner à MongoDB les moyens d'établir une répartition aussi équitable que possible. Une répartition sur la date de création seule est possible, et suffisante car les articles seront logiquement ajoutés régulièrement au fil du temps. Cependant, dans le cas d'une importation en masse d'articles qui porteraient tous la même date, MongoDB n'aurait d'autre choix que de charger une partition au détriment des autres.

Nous utilisons cette clé pour répartir une collection à l'aide de la commande `shardcollection` :

```
use admin
db.runCommand( { shardcollection: "passerelles.articles", key: { "date_création": 1,
"catégories": 1 } } );
```

La configuration est terminée, MongoDB s'occupera ensuite de répartir automatiquement les données au mieux entre les shards. La configuration pourra être vérifiée dans les collections de la base de données `config`, qui contient toutes les métadonnées de la répartition :

```
mongos> use config
mongos> show collections
mongos> db.collections.find()
{ "_id": "passerelles.articles", "lastmod": ISODate("1970-01-16T10:32:54.791Z"),
"dropped": false, "key": { "date_création": 1, "catégories": 1 }, "unique": false }
```

Voilà pour ce qui concerne notre exploration de MongoDB.

9

Riak

Riak est un moteur NoSQL décentralisé, fondé sur les principes de Dynamo d'Amazon, qui offre d'excellentes performances. Il a été créé et développé par la société Basho.

La grande force de Riak est sa capacité à monter en charge simplement, en ajoutant de nouvelles machines au cluster. Riak distribue automatiquement les données entre les machines du cluster.

Mise en œuvre

Installation

Riak est disponible en téléchargement sur le site de la société Basho, à l'adresse suivante : http://wiki.basho.com/Installation.html. Les étapes de l'installation sont clairement détaillées par environnement. Nous présentons ici la méthode adaptée à Debian et Ubuntu.

Installation du paquet

Avant toute chose, il convient d'installer une version spécifique de la bibliothèque SSL. À l'heure où nous écrivons ces lignes, Riak utilise la version 0.9.8 de la bibliothèque `libssl`. Les versions récentes de nos distributions intègrent au moins la version 1 de `libssl`. Nous pouvons le vérifier en effectuant une recherche dans les paquets grâce à la commande suivante :

```
sudo dpkg -S libssl
```

Il nous faut donc installer une version antérieure. Ceci peut être fait sans danger, car plusieurs versions de `libssl` peuvent être installées sur le même système.

```
sudo apt-get install libssl0.9.8
```

Ensuite, nous téléchargeons Riak à l'adresse http://wiki.basho.com/Installing-on-Debian-and-Ubuntu.html et nous l'installons.

```
wget http://downloads.basho.com.s3-website-us-east-1.amazonaws.com/riak/CURRENT/
ubuntu/precise/riak_1.2.1-1_amd64.deb
```

Puis nous installons le paquet :

```
sudo dpkg -i riak_1.2.1-1_amd64.deb
```

Installation des sources

Pour notre exemple, nous avons effectué l'installation à partir des sources, le paquet n'étant pas encore disponible. Pour ce faire, nous avons d'abord installé les paquets nécessaires à la compilation et à Erlang :

```
sudo apt-get install build-essential libc6-dev git libncurses5-dev openssl libssl-dev
wget http://erlang.org/download/otp_src_R15B01.tar.gz
tar zxvf otp_src_R15B01.tar.gz
cd otp_src_R15B01
./configure && make && sudo make install
```

Puis nous avons téléchargé et compilé les sources de Riak :

```
wget http://downloads.basho.com.s3-website-us-east-1.amazonaws.com/riak/CURRENT/riak-
1.2.1.tar.gz
tar zxvf riak-1.2.1.tar.gz
cd riak-1.2.1
make rel
```

Si le `make` se termine sans erreur, Riak compilé sera disponible dans le sous-répertoire `rel` du répertoire dans lequel vous avez lancé le `make`. Vous pouvez ensuite le copier dans un répertoire adéquat de votre système.

```
sudo mv rel /usr/local/riak-1.2.1
cd /usr/local/riak-1.2.1/riak/bin
./riak start
./riak ping
pong
```

La commande `riak ping` permet de déterminer simplement si le serveur est démarré. Un `pong` est retourné en cas de succès.

Configuration

Les fichiers de configuration de Riak se trouvent dans le répertoire `etc/`. Pour notre installation exemple, il s'agit de `/usr/local/riak-1.2.0/riak/etc`. Dans une installation binaire, vous les trouverez plus probablement dans `/etc/riak`. La configuration générale se trouve dans le fichier `app.config`. Les deux principales sections sont `riak_api`, qui définit les options pour accéder à Riak via Protocol Buffers, et `riak_core`, qui définit la configuration générale et l'accès par HTTP. Pour ouvrir l'accès à l'interface REST de Riak depuis des machines clientes, nous allons modifier le code :

```
{riak_core, [
        …
        {http, [ {"127.0.0.1", 8098 } ]},
        …
```

par :

```
{http, [ {"0.0.0.0", 8098 } ]},
```

Nous relançons ensuite Riak :

```
cd /usr/local/riak-1.2.0/riak/bin
./riak start
```

Un accès à Riak via un navigateur web retourne la liste représentée à la figure 9-1, il s'agit de la liste des ressources REST à disposition.

Figure 9-1
Liste des ressources REST à disposition

Utilisation de l'API REST

Accédons à Riak en utilisant curl. Pour tester l'accès à Riak, nous pouvons utiliser la commande `ping` via l'interface REST :

```
curl -v http://localhost:8098/ping
```

Nous recevons la réponse :

```
* About to connect() to localhost port 8098 (#0)
*   Trying 127.0.0.1... connected
* Connected to localhost (127.0.0.1) port 8098 (#0)
> GET /ping HTTP/1.1
> User-Agent: curl/7.21.6 (i686-pc-linux-gnu) libcurl/7.21.6 OpenSSL/1.0.0e
zlib/1.2.3.4 libidn/1.22 librtmp/2.3
> Host: localhost:8098
> Accept: */*
>
< HTTP/1.1 200 O
< Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted it blue)
< Date: Tue, 13 Nov 2012 11:03:56 GM
< Content-Type: text/html
< Content-Length:
<
* Connection #0 to host localhost left intact
* Closing connection #0
OK
```

L'équivalent Riak d'une table contenant des données s'appelle un bucket. Il est inutile de créer un bucket, sa création sera automatique lors de la première insertion de données. Nous allons insérer notre article de blog du site Passerelles :

```
curl -X PUT -H «Content-Type: application/json» -d '
{
«auteur»:
{
    «prénom»:»Annie»,
    «nom»:»Brizard»,
    «e-mail»:»annie.brizard@cocomail.com»
},
«titre»:»pourquoi les éléphants ont-ils de grandes oreilles?»
}' http://localhost:8098/buckets/passerelles/keys/1?returnbody=true
```

Comme nous l'avons demandé par (`returnbody=true`), nous recevons le document en retour.

Interrogeons maintenant notre bucket :

```
curl -X GET http://localhost:8098/buckets/passerelles/keys?keys=true
```

Nous recevons la liste des clés, en format JSON :

```
{«keys»:[«1»]}
```

Il est possible de recevoir la liste des clés dans un flux pour permettre au client de le traiter par paquets :

```
curl -X GET http://localhost:8098/buckets/passerelles/keys?keys=stream
```

Retrouvons maintenant le document lui-même :

```
curl -X GET http://localhost:8098/buckets/passerelles/keys/1
```

Puis supprimons-le:

```
curl -X DELETE http://localhost:8098/buckets/passerelles/keys/1
```

Programmation client

Nous allons utiliser Riak avec Python en utilisant le client Python développé par Basho.

```
./python_env/nosql/bin/pip install riak
```

En interne, ce client utilise l'interface REST ou protobuf. Nous allons bien sûr préférer protobuf pour ses meilleures performances. Voici un premier morceau de code qui nous connecte au serveur, insère notre article sur le site Passerelles, puis récupère la liste des clés du bucket `passerelles`.

```
#!/usr/bin/python
# coding=utf-8

import riak
import json

cli = riak.RiakClient(host=»192.168.0.22», port=8087, transport_class=riak.
RiakPbcTransport)
pas = cli.bucket('passerelles')

article = pas.new('1', data={
    «auteur»:
    {
        «prénom»:»Annie»,
        «nom»:»Brizard»,
        «e-mail»:»annie.brizard@cocomail.com»
    },
    «titre»:»pourquoi les éléphants ont-ils de grandes oreilles?»
})

article.store()

print pas.get_keys()
```

Le retour est une liste de clés:

```
['1']
```

Nous pouvons obtenir le document de la façon suivante:

```
article = pas.get('1').get_data()
```

La méthode `get()` retourne un objet Riak, et la méthode `get_data()` de cet objet retourne un dictionnaire Python. Nous générons donc directement un dictionnaire.

Administration

Riak est très simple à administrer. Il s'agit après tout d'un dépôt de paires clé-valeur dans la lignée de Dynamo, offrant un serveur largement autoconfiguré aux fonctionnalités simples et efficaces. Comme Riak est par nature un système distribué, nous allons voir comment gérer les nœuds du cluster.

Configuration du nœud

La configuration du premier nœud implique de modifier les fichiers de configuration et d'utiliser l'outil `riak-admin` pour créer un cluster. Premièrement, nous devons stopper Riak, sinon on ne pourra plus y accéder après changement des fichiers de configuration.

```
sudo ./riak stop
```

Nous fixons l'IP sur laquelle écoute Riak pour indiquer une IP liée à une interface de notre serveur. Nous allons le faire aussi bien pour Protocol Buffers (section `riak_api`) que pour l'interface HTTP :

```
{riak_api, [
        {pb_ip,   «192.168.0.22»},
        …
}
{riak_core, [
        …
        {http, [{«192.168.0.22», 8098}]},
        …
```

Puis nous modifions également le nom au niveau de la machine virtuelle Erlang, dans le fichier `vm.args`. Nous changeons la ligne :

```
-name riak@127.0.0.1
```

en :

```
-name riak@192.168.0.22
```

Nous redémarrons ensuite Riak :

```
sudo ./riak start
```

Si la tentative de démarrage du nœud provoque une erreur, vous pouvez obtenir plus d'informations en activant les messages de la console Erlang :

```
sudo ./riak console
rm -rf /var/lib/riak/ring/*
```

La première machine va être automatiquement membre du cluster, vous aurez à configurer les autres machines pour joindre ce cluster. Pour obtenir une liste des membres du cluster, utilisez l'outil `riak-admin` de la façon suivante :

```
./riak-admin status | grep ring_members
```

Dans notre cas, avec une seule machine, le retour est :

```
ring_members: ['riak@192.168.0.22']
```

Pour ajouter une nouvelle machine au cluster, configurez le nœud, puis rejoignez le cluster à l'aide de `riak-admin` :

```
sudo ./riak-admin cluster join riak@192.168.0.22
sudo ./riak-admin cluster plan
sudo ./riak-admin cluster commit
```

Voilà pour ce qui concerne notre exploration de Riak.

10

Redis

Redis est un moteur de base de données en mémoire (on pourrait l'appeler un serveur de structure de données) développé à la base par un programmeur italien nommé Salvatore Sanfilippo. Son développement à partir de 2009 a été très rapide et sa popularité fut foudroyante. En 2010, VMWare recruta Salvatore, puis Pieter Noordhuis, un contributeur important de Redis, pour travailler à plein temps sur le développement du produit, tout en le laissant en licence libre (BSD).

Présentation

Redis est un moteur non relationnel qui travaille principalement en mémoire. Son objectif est de manipuler le plus rapidement possible les structures de données qui y sont maintenues. Il est à la fois un outil de gestion de paires clé-valeur et un cache de données à la manière de memcached. En effet, il offre les mêmes fonctionnalités et avantages que memcached, mais avec un modèle beaucoup plus riche et solide. Le but de Redis est d'être très rapide, très efficace et léger. Et il remplit ses objectifs à la perfection. C'est de plus un outil avec lequel il est très agréable de travailler. En général, quand on l'essaie, on l'adopte.

memcached

memcached est un système de cache développé à l'origine par les développeurs de LiveJournal, un outil de création de blogs, très utilisé dans les développements web par beaucoup de grands acteurs comme Twitter et Facebook, ou des sites comme Wikipédia. Les structures de données manipulées par memcached sont très simples : il s'agit de paires clé-valeur, donc des tableaux de hachage, aussi appelés *maps*. memcached n'offre aucune persistance.

Les types de données

Redis ne manipule pas de documents au sens où on l'a entendu jusqu'à présent, par exemple des structures JSON. Il travaille en paires clé-valeur, les valeurs pouvant être de cinq types, détaillées dans les sections suivantes.

Chaînes

Une chaîne de caractères en Redis est un peu plus qu'une chaîne. Elle permet également de stocker correctement les valeurs numériques et binaires *(binary safe)*, ce qui la rend adaptée pour les calculs ou les fichiers binaires comme des images. La taille maximale supportée par ce type est de 512 Mo. Redis offre nativement un certain nombre de commandes qui permettent de manipuler les chaînes et de les utiliser de diverses façons, par exemple en modifiant la valeur numérique qu'elles contiennent avec des commandes comme `INCR`, `INCRBY` ou `DECR` et `DECRBY`.

Listes

Les listes sont simplement des listes de chaînes – chaînes au sens large de Redis comme nous venons de le voir dans la section précédente – maintenues dans un ordre défini à l'insertion. Lorsque vous ajoutez une chaîne à la liste, vous pouvez définir si vous voulez l'ajouter au début (à gauche) ou à la fin (à droite) de la liste, à l'aide des commandes `LPUSH` (gauche) ou `RPUSH` (droite).

Ensembles

Un ensemble *(set)* est une collection non triée de chaînes. Il représente un ensemble correspondant plus ou moins à ce que la théorie des ensembles entend par un set : il est non trié et n'accepte pas les doublons. Si vous insérez une valeur déjà présente, cela n'aura aucun effet. L'accès à un élément de l'ensemble offre d'excellentes performances même si l'ensemble contient un grand nombre d'éléments. Le nombre d'éléments est limité à un entier 32 bits non signé, c'est-à-dire $2^{32} - 1$. Quelques opérateurs ensemblistes sont disponibles au niveau du serveur pour effectuer des opérations très rapidement : union, intersection et différence.

Hachages

Ils représentent une table de hachage (un dictionnaire), ce qui permet de stocker une représentation d'objet, à l'image d'un document JSON plat (non hiérarchique), dans Redis. Le stockage est optimisé, ce qui fait que le hachage prend peu de place en mémoire.

Ensembles triés

L'expression « ensemble trié » sonne un peu comme un oxymore, mais le résultat fait sens. L'ensemble trié se comporte comme un ensemble mais chaque membre est en plus associé à une note (un score) qui permet de trier l'ensemble. Plusieurs membres peuvent avoir la même note. La manipulation et le tri sont très rapides. Il est ceci dit inutile de mentionner ce genre de choses : tout est très rapide en Redis. Avec un ensemble trié, vous pouvez gérer avec une grande efficacité tout type d'ordre ou de score, du genre jeux, tops ou best of.

Mise en œuvre

Redis est une perle : efficacité, élégance et simplicité. Tout ce que nous allons voir dans les sections suivantes sera donc très facile à implémenter.

Installation

L'installation de Redis est très simple. Tout d'abord, téléchargez la dernière version disponible du moteur à l'adresse suivante : http://redis.io/download. Nous avons téléchargé la version 2.6.0 release candidate 5 qui était la dernière version au moment de la rédaction de ce chapitre. Nous l'avons ensuite décompressée et nous avons compilé le moteur.

```
wget http://redis.googlecode.com/files/redis-2.6.0-rc7.tar.gz
tar xvzf redis-2.6.0-rc7.tar.gz
cd redis-2.6.0-rc7
make
```

Redis est développé en C, il est donc nécessaire d'installer au préalable un compilateur C comme GCC pour que la compilation se déroule correctement.

Redis est également disponible sous forme de paquets dans plusieurs distributions. Sur Debian et Ubuntu, le paquet s'appelle `redis-server`, il est disponible dans les dépôts officiels. Vous pouvez l'installer ainsi :

```
sudo apt-get install redis-server
```

Cette installation par paquet à l'avantage de placer Redis dans les bons répertoires sur votre distribution. En revanche, vous n'obtiendrez pas la dernière version du moteur. Compte tenu de la simplicité de compilation et d'installation de Redis, nous vous conseillons de réaliser une installation manuelle. Après compilation, vous pouvez déplacer manuellement les quelques exécutables nécessaires à Redis :

```
sudo cp redis-server redis-cli redis-benchmark /usr/local/bin
```

Pour démarrer le serveur Redis, vous pouvez simplement invoquer l'exécutable `redis-server`. Les quelques paramètres disponibles permettent d'indiquer un port sur lequel écouter (le port par défaut est le 6379), une relation de serveur secondaire par rapport à un serveur maître, ou l'emplacement du fichier de configuration à prendre en compte. Voici des exemples de lancement du serveur Redis :

```
nohup redis-server /etc/redis/redis.conf --loglevel verbose &
nohup redis-server --port 7777 --slaveof 127.0.0.1 8888 &
```

Pour s'assurer que Redis est lancé, lancez un `ping` à l'aide du client `redis-cli`. Il devrait vous renvoyer un `PONG`.

```
>redis-cli ping
PONG
```

Configuration

Si vous avez besoin d'ajuster la configuration de Redis, un fichier de configuration peut être créé et indiqué au démarrage du serveur. Un fichier d'exemple (`redis.conf`) est disponible dans les sources, à cette adresse : https://github.com/antirez/redis. Nous verrons plus en détail certains éléments de la configuration plus tard, attardons-nous déjà sur les plus évidents. Voici un exemple de fichier de configuration simple :

```
daemonize yes
port 6379
bind 127.0.0.1
timeout  120
loglevel notice
logfile s/var/log/redis.log
syslog-enabled no
databases 16
```

Par défaut, Redis ne tourne pas en tant que démon, c'est la raison pour laquelle dans notre exemple de lancement de `redis-server`, nous avons lancé le processus en arrière-plan à l'aide du caractère `&`. En indiquant `daemonize yes`, nous pouvons faire en sorte que Redis démarre en tant que démon. Le port et l'IP sur lesquels Redis écoute sont ensuite définis. 6379 est le port par défaut, et si l'option `bind` n'est pas indiquée, Redis répondra sur toutes les interfaces. L'option `timeout` permet de fermer une connexion inactive, ici après 120 secondes. La valeur par défaut est 0, pas de timeout. Ensuite, nous trouvons quelques options de journalisation de l'application : ici nous définissons le niveau de journalisation à `notice`, qui est un niveau adapté pour la production, et nous indiquons le fichier de journal. Une autre option est d'envoyer les informations à `stdout`. Nous décidons de ne pas les traiter par `syslog`. Finalement, nous indiquons le nombre de bases de données : 16. En Redis, une base de données est simplement un espace de stockage de paires clé-valeur en mémoire identifié par un numéro. La base de données par défaut est 0.

Utilisation de redis-cli

Redis est livré avec une invite interactive nommée `redis-cli`, qui permet d'envoyer des commandes au serveur. Utilisons-la pour stocker notre entrée de blog Passerelles :

```
redis-cli
redis 127.0.0.1:6379> SELECT 1
OK
redis 127.0.0.1:6379[1]>HMSET article:1 prenom «Annie» nom «Brizard» e-mail «annie.
brizard@cocomail.com» titre «Ce sont les girafes qui ont de grandes oreilles, non?»
OK
redis 127.0.0.1:6379[1]> SADD tag:animaux 1
(integer) 1
redis 127.0.0.1:6379[1]> SADD tag:questions 1
(integer) 1
redis 127.0.0.1:6379[1]> SADD article:1:tags animaux questions
(integer) 2
redis 127.0.0.1:6379[1]> SINTER tag:animaux tag:questions
```

```
1) "1"
redis 127.0.0.1:6379[1]> HVALS article:1
1) «Annie»
2) «Brizard»
3) «annie.brizard@cocomail.com»
4) «Ce sont les girafes qui ont de grandes oreilles, non?»
redis 127.0.0.1:6379[1]> HGET article:1 e-mail
«annie.brizard@cocomail.com»
redis 127.0.0.1:6379[1]> MULTI
OK
redis 127.0.0.1:6379[1]> RENAMENX article:1 article:2
QUEUED
redis 127.0.0.1:6379[1]> SREM tag:animaux 1
QUEUED
redis 127.0.0.1:6379[1]> SREM tag:questions 1
QUEUED
redis 127.0.0.1:6379[1]> SADD tag:animaux 2
QUEUED
redis 127.0.0.1:6379[1]> SADD tag:questions 2
QUEUED
redis 127.0.0.1:6379[1]> EXEC
1) (integer) 1
2) (integer) 1
3) (integer) 1
4) (integer) 1
5) (integer) 1
redis 127.0.0.1:6379[1]>
redis 127.0.0.1:6379[1]> TYPE article:2
hash
```

Comme vous le voyez, Redis comporte un ensemble de commandes très simples, qui permettent principalement de manipuler les types de données disponibles. La liste des commandes est disponible à l'adresse suivante : http://redis.io/commands.

Voyons rapidement les commandes utilisées.

- `SELECT` – Change le contexte de la base de données. Ici, nous joignons la base de données 1.
- `HMSET` – Attribue des valeurs à un hachage. Ici, nous stockons notre article `article:1`.
- `SADD` – Ajoute un ou plusieurs membres à un ensemble. Si la clé n'existe pas, elle sera créée, sinon le membre sera ajouté à la clé existante. Nous l'utilisons ici pour ajouter des tags à notre article : nous maintenons des tags sous forme d'ensembles dont la clé est `tag:<nom_du_tag>`, et nous ajoutons les identifiants d'articles qui portent ce tag comme membres de l'ensemble.
- Pour référence, nous créons aussi un ensemble `article:1:tags` pour lister les tags de l'article. Il sera plus simple de retrouver les tags à partir d'un article de cette manière.
- `SINTER` – Effectue une intersection entre deux ensembles. Ici, nous voulons savoir quels sont les identifiants d'articles présents à la fois dans `tag:animaux` et `tag:questions`. Redis nous retourne `"1"`.
- `HVALS` – Retourne la table de hachage complète contenue dans une clé.

- `HGET` – Retourne un élément d'une table de hachage. Ici, nous retrouvons l'e-mail de l'auteur de l'article 1.
- `MULTI` – Commence une transaction. Toutes les instructions qui suivent seront maintenues dans une file d'attente et ne seront exécutées qu'à l'appel de la commande `EXEC`. Le résultat des commandes suivantes sera donc `QUEUED`. Pour annuler la transaction, utilisez la commande `DISCARD`. Par transaction, il faut simplement entendre une file d'attente. Rien n'est exécuté avec l'`EXEC`, même pas les commandes de lecture. Nous maintenons cette transaction car nous avons décidé de changer la clé de l'article.
- `RENAMENX` – Renomme une clé.
- `SREM` – Supprime un membre dans un ensemble.
- `EXEC` – Valide la transaction et exécute toutes les commandes dans la file.
- `TYPE` – Retourne le type de donnée contenu dans la clé. Ici, un hash.

Exemples d'applications clientes

Nous allons développer une application cliente en Python. Comme vous en avez maintenant l'habitude, nous allons installer le client Redis pour Python (http://pypi.python.org/pypi/redis/).

```
~/python_env/nosql/bin/pip install redis
```

Au lieu d'utiliser l'exemple traditionnel de notre blog Passerelles, nous allons réaliser un code utilisable par tous, petit par la taille mais grand par la puissance. En quatre lignes de code (sans compter l'importation des modules et une ligne de débogage), le script Python suivant suit le flux public de Twitter et récupère des tweets concernant quelques mots-clés, ici `"nosql"`, `"sql"` et `"redis"`.

```
#!/usr/bin/python

import redis
import tweetstream

words = ["nosql", "sql", "redis"]

r = redis.StrictRedis(host='localhost', port=6379, db=2)
stream = tweetstream.FilterStream("user", "password", track=words)

for tweet in stream:
    if tweet.has_key("text"):
        print tweet['user']['screen_name'] + ": " + tweet['text'] , "\n"
        r.set(tweet['id'], tweet['text'])
```

Nous créons d'abord une liste de mots-clés à surveiller, puis nous ouvrons une connexion à Redis en instanciant la classe `redis.StrictRedis`. Cette classe s'appelle `strict` car elle tente autant que possible de respecter la syntaxe « officielle » des commandes Redis. Nous passons les informations de connexion au constructeur.

Pour suivre le flux Twitter, nous instancions la classe `tweetstream.FilterStream` (le module `tweetstream` est documenté à l'adresse suivante : http://pypi.python.org/pypi/tweetstream). Nous bouclons ensuite dans les tweets du flux, et comme il s'agit d'un flux, le contact restera ouvert et la boucle continuera à attendre de nouveaux tweets tant que le script ne sera pas tué.

Pour cet exemple de code, nous avons ajouté une ligne de débogage qui réalise un `print` du tweet repéré. Ensuite, la ligne qui insère le tweet dans un string est la suivante :

```
r.set(tweet['id'], tweet['text'])
```

Nous utilisons l'identifiant Twitter du tweet comme clé. Ce n'est pas forcément une très bonne idée, nous pourrions plutôt utiliser une combinaison date-auteur, par exemple, mais cela suffira pour notre exemple.

Nous avons laissé tourner ce code quelques jours. En utilisant le client Redis, nous pouvons tester la taille de la base, c'est-à-dire le nombre de clés, à l'aide de la commande `DBSIZE`, qui retourne 15171. La méthode `dbsize()` de l'objet `StrictRedis` en Python retourne la même valeur. Voici du code Python qui affiche la taille de la base de données et retrouve tous les tweets.

```
#!/usr/bin/python

import redis

r = redis.StrictRedis(host='localhost', port=6379, db=0)
print r.dbsize()

for key in r.keys('*'):
        print r.get(key)
```

Nous utilisons ici la méthode `keys()` qui retourne une liste de clés correspondant à un pattern de recherche, ici `'*'` pour toutes les clés, puis nous bouclons et affichons chaque élément à l'aide de la méthode `get()`. Un fragment de retour dans la console est affiché ci-dessous :

```
Know anyone for this job? SQL Report Developer  in Naperville, IL http://t.co/R8GqJwiV
T-SQL MERGE Statement Tips http://t.co/eQBcmKSb by @ItzikBenGan @sqlservermag
#SQLServer
RT @couchbase: The scalability &amp;amp; performance needs of today's apps differs
dramatically from the past. New whitepaper on why #NoSQL: ...
#Jobs #jobsite Graduate Consultant - Leading Software Company (Java, XML, SQL)
http://t.co/YoAwxhy4 #Fishajobs
#SQL #Database #Jobs Sql Database Administrator at Unc Charlotte (Charlotte, NC)
http://t.co/jRznyw4h
SQL Back End Lead /Developer - 6 Month Rolling Contract - http://t.co/HTQoaUra
 #jobs #london #dev #it
@IwilBeThere Lee tutoriales SQL Injected, mira videos si queres jaja
#Job, Oracle BPM Developer with:CSS,HTML,Java,Oracle,SOA,SQL http://t.co/iqgoHhEl
BlackNova Traders SQL Injection: BlackNova Traders, a web-based game similar to the
BBS game TradeWars, suffers ... http://t.co/E6cL2GFt
```

Voici brièvement un autre exemple de code Python, stockant un hachage. Le pilote Python convertit automatiquement un dictionnaire Python en table de hachage Redis, pour le stocker à l'aide de la méthode `hmset()`, ou le restituer à l'aide de la méthode `hgetall()`.

```
#!/usr/bin/python

import redis

r = redis.StrictRedis(host='localhost', port=6379, db=1)

r.hmset('contact:ajar', dict(
                prenom='Emile',
                nom='Ajar',
                Profession='Ecrivain'
        )
)

dic = r.hgetall('contact:ajar')
print dic
```

Pub/sub

Redis permet aussi de gérer des canaux d'information, correspondant au pattern de messagerie `publish-subscribe` : un client s'abonne à un canal, il peut ainsi y envoyer des publications et recevoir les messages des autres abonnés. Les commandes de Redis permettant de gérer ces canaux sont très simples : `SUBSCRIBE`, `UNSUBSCRIBE` et `PUBLISH`. Nous allons transformer notre code de suivi de flux Twitter pour envoyer un message sur un canal et avertir d'autres clients Redis. Le code du « serveur » de message est le suivant :

```
#!/usr/bin/python

import redis
import tweetstream

words = ["nosql", "sql", "redis"]

r = redis.StrictRedis(host='localhost', port=6379, db=1)
stream = tweetstream.FilterStream("user", "password", track=words)

for tweet in stream:
        if tweet.has_key("text"):
                r.publish('mytwitter', tweet['text'])
```

Nous reprenons l'essentiel du code. La seule différence est l'action réalisée vers Redis à la réception d'un tweet : nous effectuons simplement un `publish()` sur le canal `mytwitter` avec le corps du tweet. Nous exécutons le script, qui va tourner jusqu'à interruption manuelle. Nous lançons ensuite le script suivant :

```
#!/usr/bin/python
```

```
import redis
import tweetstream

words = ["nosql", "sql", "redis"]

r = redis.StrictRedis(host='localhost', port=6379, db=1)
ps = r.pubsub()
ps.subscribe("mytwitter")

for msg in ps.listen():
    print msg['data']
```

Il s'agit bien sûr du client. Un objet `pubsub` est instancié et nous appelons la méthode `subscribe()` pour l'abonner à notre canal. Il suffit ensuite de boucler sur la méthode `listen()` pour récupérer petit à petit les publications de notre autre script. Voici un extrait du résultat, apparaissant au fur et à mesure de l'arrivée des tweets:

```
@ToonyHouusni oulalalaa jsais pas jte redis après atta
SQL Server Reporting Analyst: Robert Half Technology is looking for a SQL Server
Reporting Analyst for... http://t.co/jaFIdCri #job #rht
@BelieberBoyPOL  bo ja chodze do sql.xd ps.w piðtek mnie nie bðdzie.:D
HTML table porting to SQL: Is there anyway i can extract data from a HTML table (i
tried jsoup) and then save it... http://t.co/GBu9mbYb
@prakharprasad @yashkadakia That's not SQL.:)
#SQL #job - Correction and optimization of a Mobile Site - Los Angles area ... ($20 -
30/hr) - http://t.co/jXol6m35 #jobs
```

Maintenance

Redis est un système très simple. Conçu au départ comme un moteur totalement en mémoire, il intègre petit à petit plus de fonctionnalités, présentées rapidement dans les sections suivantes.

Performances

Un test de performances de Redis peut être simplement effectué avec l'outil `redis-benchmark`, qui effectue une suite d'opérations de tests et retourne des statistiques de performances pour les opérations les plus courantes. Voici un extrait des résultats de son appel sur notre système:

```
====== LRANGE_600 (first 600 elements) ======
  10000 requests completed in 2.04 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

0.01% <= 2 milliseconds
0.16% <= 3 milliseconds
0.99% <= 4 milliseconds
52.87% <= 5 milliseconds
```

```
95.09% <= 6 milliseconds
97.93% <= 7 milliseconds
99.05% <= 8 milliseconds
99.86% <= 9 milliseconds
99.96% <= 10 milliseconds
100.00% <= 10 milliseconds
4909.18 requests per second

====== MSET (10 keys) ======
  10000 requests completed in 0.34 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

6.88% <= 1 milliseconds
92.90% <= 2 milliseconds
97.85% <= 3 milliseconds
99.51% <= 5 milliseconds
99.76% <= 7 milliseconds
100.00% <= 7 milliseconds
29850.75 requests per second
```

Comme vous pouvez le constater, chaque opération est testée et les résultats vous permettent déjà de juger de l'efficacité de Redis sur votre système. La plupart des opérations de Redis offrent une vitesse relativement constante même avec une base de données dont le volume augmente.

INFO et MONITOR

La commande Redis `INFO` donne aussi quelques indications sur le système. Voici une partie du résultat sur notre système :

```
# Memory
used_memory:4350456
used_memory_human:4.15M
used_memory_rss:7872512
used_memory_peak:7173840
used_memory_peak_human:6.84M

# CPU
used_cpu_sys:54.75
used_cpu_user:25.59
used_cpu_sys_children:0.03
used_cpu_user_children:0.11
```

Nous y voyons l'occupation mémoire actuelle et le temps processeur consommé. Une explication sur les compteurs est disponible à cette adresse : http://redis.io/commands/info.

La commande `MONITOR` permet de tracer toutes les commandes envoyées au serveur.

slowlog

Un journal des requêtes lentes *(slowlog)* est aussi maintenu en mémoire par Redis. Par requêtes lentes, on entend les requêtes qui dépassent le temps spécifié en microsecondes par l'option de configuration `slowlog-log-slower-than`. L'option `slowlog-max-len` permet d'indiquer un nombre maximal d'enregistrements à conserver dans le slowlog. Vous pouvez ensuite l'interroger avec la commande `slowlog`, par exemple :

```
slowlog get
slowlog get 10
slowlog len
slowlog reset
```

La première commande récupère la liste des commandes enregistrées dans le slowlog. La suivante en retourne seulement les dix dernières (les dix plus récentes). `slowlog len` retourne le nombre d'entrées et `slowlog reset` vide le slowlog.

Persistance

Redis est par nature une base de données en mémoire. Un système de persistance sur disque y a été ajouté, lequel sauvegarde l'état des bases de données jusqu'au prochain redémarrage de Redis. Le système par défaut est nommé RDB, qui se base sur la prise régulière ou manuelle d'un *snapshot*, nommé `dump.rdb`. Ce snapshot est stocké sur le disque dans le répertoire courant de Redis (celui dans lequel il a démarré) ou dans un répertoire indiqué dans le fichier de configuration si celui-ci est spécifié au démarrage. Par exemple, notre `dump.rdb` contenant nos 15 000 tweets pèse un peu moins de 2 Mo.

L'option de configuration `save` permet d'indiquer la fréquence de sauvegardes RDB. Nous pouvons l'inscrire dans un fichier de configuration ou la modifier avec la commande `CONFIG SET`. La commande `CONFIG GET` retrouve la valeur actuelle. Pour notre exemple, cela donne :

```
redis-cli config get save
1) "save"
2) "3600 1 300 100 60 10000"
```

ce qui signifie : une sauvegarde doit être faite après 3 600 secondes s'il y a au moins une modification de données, après 300 secondes s'il y a au moins 100 modifications, et après 60 secondes s'il y a au moins 10000 modifications. La commande `BGSAVE` permet de lancer une sauvegarde manuelle non bloquante (BG = *background*). Lorsque Redis redémarre, le fichier `dump.rdb` est automatiquement rechargé pour remplir les bases de données Redis.

Cette méthode de sauvegarde d'instantanés peut bien sûr provoquer des pertes de données si le serveur est arrêté brusquement : les données modifiées depuis la dernière sauvegarde seront perdues. Une autre méthode de persistance est disponible, nommée AOF (*Append-Only File*, fichier à ajout seulement), qui équivaut plus ou moins à un *Write-Ahead Log*. Pour activer AOF, assurez-vous que votre fichier de configuration contienne la ligne suivante :

```
appendonly yes
```

Chaque modification de données sera alors ajoutée au journal, lequel sera rejoué au démarrage de Redis. Un fichier AOF sera plus volumineux qu'un fichier RDB, et le démarrage sera plus lent, mais il n'y aura pas de perte de données. Vous pouvez améliorer les performances d'AOF grâce à l'option de configuration `appendfsync` qui vous permet de configurer plus finement les écritures de journal. Les deux valeurs utiles sont reprises dans le tableau 10-1.

Tableau 10-1. Valeurs pour appendfsync

Valeur	Description
`always`	Ajoute au fichier AOF à chaque modification, plus sûr mais plus intensif.
`everysec`	Ajoute au fichier AOF à chaque seconde. Relativement sûr et plus léger.

Si le fichier AOF est un journal, et qu'il est rejoué à chaque démarrage, vous devriez maintenant vous dire que c'est une option particulièrement inefficace : au fur et à mesure des modifications, le fichier AOF devrait prendre une taille respectable, et le démarrage devrait devenir de plus en plus long. Heureusement, Redis va compacter régulièrement ce fichier, en éliminant toutes les étapes intermédiaires. Si une clé a été modifiée cent fois, seule la dernière valeur est conservée afin de reconstruire la donnée la plus fraîche. Ce compactage est exécuté automatiquement en tâche de fond, mais vous pouvez aussi le déclencher manuellement avec la commande `BGREWRITEAOF`.

Réplication

Un serveur Redis peut être configuré pour être le secondaire d'un autre serveur, dans une architecture maître-esclave arborescente (un secondaire peut se baser sur un secondaire). Pour cela, il suffit de mentionner dans le fichier de configuration :

```
slaveof ip port
```

en indiquant l'IP et le port du serveur maître. Le serveur secondaire est la copie exacte du serveur maître. L'option `slave-serve-stale-data` indique le comportement à observer si la connexion avec le maître est perdue. La valeur par défaut est `yes` : le serveur continue à répondre avec des données potentiellement désynchronisées. Si l'option vaut `no`, une requête sur le serveur secondaire retourne une erreur « SYNC with master in progress ».

La commande `SLAVEOF` permet de déterminer si le serveur est un secondaire.

L'option `slave-read-only`, par défaut à `yes`, indique si les écritures sont interdites sur le secondaire. Écrire sur un Redis secondaire n'a d'intérêt que pour des données éphémères : il n'y a pas de synchronisation vers le serveur maître et les données pourront donc être écrasées par des modifications effectuées sur le maître.

Redis-sentinel et Redis Cluster

Redis-sentinel (http://redis.io/topics/sentinel) est un outil disponible depuis longtemp en bêta, mais utilisé très souvent en production parce que parfaitement stable. Il s'agit d'un outil de monitoring et de gestion de haute disponibilité. Il permet, conjointement à l'option de configuration `slave_priority`, de promouvoir automatiquement un secondaire en maître si ce dernier est indisponible.

La version 3 de Redis intègre redis-sentinel et l'améliore en un système de sharding intégré qui permet de transformer Redis en véritable moteur distribué.

Redis Cluster (http://redis.io/topics/cluster-tutorial) est un système décentralisé où les nœuds communiquent à l'aide un canal TCP supplémentaire. Les données sont shardées automatiquement par Redis en utilisant la technique des vnodes, nommées ici des *hash slots*. Le nombre de slots est fixé à 16 384, et le placement de la clé est simplement effectué en calculant le modulo 16 384 du hash de la clé.

La réplication est assurée selon un modèle maître-secondaire. Si vous voulez créer un cluster de trois shards, vous ajoutez trois machines pour accueillir les réplicas. Vous devez avoir plus de deux maîtres, car le basculement automatique se fait par quorum : il faut donc être au moins deux pour prendre la décision.

Les ajouts à la configuration de Redis pour Redis Cluster sont les suivants :

```
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

Avec `cluster-enabled`, on indique bien entendu que Redis est en cluster. On définit un fichier de configuration du cluster qui est créé et alimenté automatiquement par Redis pour maintenir les informations de l'état du cluster.

La valeur `cluster-node-timeout` configure en millisecondes combien de temps les autres masters vont attendre avant de considérer qu'un nœud est tombé s'ils n'arrivent pas à le joindre.

Nous avons mentionné l'option `appendonly` que nous connaissons déjà pour indiquer que les nœuds de Redis Cluster doivent fonctionner en appendonly.

Lorsque les nœuds sont démarrés, depuis n'importe quelle machine, vous devez lancer un script Ruby disponible dans le répertoire des binaires de Redis, et qui s'appelle `redis-trib.rb`. Ce script va initialiser le cluster. Vous n'avez rien d'autre à faire. Voici un exemple avec six machines installées en pseudo distribué sur la même machine en écoute sur les ports TCP 7000-7005 :

```
./redis-trib.rb create --replicas 1 127.0.0.1:7000 127.0.0.1:7001 \
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005
```

Toute la question de Redis Cluster est de trouver une bibliothèque client qui le supporte. Allez sur http://redis.io/topics/cluster-tutorial pour voir l'état du développement communautaire des clients. Vous y trouvez notamment la référence au client développé par StackExchange pour .NET. StackExchange est un site important et très connu, qui utilise beaucoup Redis comme cache. Leur environnement est plutôt Microsoft et c'est pour cette raison qu'ils ont développé un client .NET.

Idéalement, il s'agit de trouver un smart client pour votre langage. Redis est capable de rediriger la commande sur le nœud qui doit héberger la clé que vous voulez écrire, mais cela nécessite un hop sur un nœud et peut-être ensuite un hop supplémentaire sur le nœud cible.

Par exemple :

```
redis 127.0.0.1:7000> set macle mavaleur
-> Redirected to slot [11182] located at 127.0.0.1:7002
OK
```

Conclusion

Redis est un merveilleux petit outil, qui devient de plus en plus intéressant avec le temps. L'intelligence de sa conception, l'élégance et la simplicité de son développement et de ses améliorations au fil des versions, en font un moteur de données très attractif. Nous vous le conseillons vivement.

11

Cassandra

Cassandra est un moteur de base de données développé à l'origine par Facebook par Avinash Lakshman, l'un des développeurs de Dynamo d'Amazon, passé depuis chez Facebook, et Prashant Malik. Le code en a été libéré en 2008.

Caractéristiques du moteur

Cassandra est un moteur de bases de données orienté colonnes. Il s'inspire donc du modèle BigTable de Google, mais emprunte également des caractéristiques à Dynamo d'Amazon. Une fois de plus, nous retrouvons les deux grands fondateurs du mouvement NoSQL. Les fonctionnalités de Cassandra sont assez complètes et son architecture est intéressante. En effet, cette dernière est décentralisée comme celle de Dynamo. Contrairement à HBase, elle ne se base pas sur un système de fichiers distribué. Cassandra est donc un moteur autosuffisant.

Modèle de données

Dans Cassandra, les données sont regroupées en familles de colonnes. Une famille de colonnes *(column family)* est plus ou moins l'équivalent d'une table dans le modèle relationnel, à la différence que son schéma n'est pas fixé à l'avance : chaque ligne peut avoir un nombre différent de colonnes. Chaque famille de colonnes est stockée dans son propre fichier sur le disque.

Une colonne est définie par son nom, qui n'est pas forcément une chaîne de caractères : le nom de colonne peut être d'un autre type, comme un entier ou un UUID, ce qui est pratique pour créer manuellement des index secondaires. La colonne contient une valeur et un horodatage (timestamp).

La version 2 de Cassandra, sortie en 2014, a un peu dissimulé ce stockage interne. Au ressenti, un développeur Cassandra se retrouve comme devant des tables de SGBDR. Nous allons en parler.

Stockage

Les écritures de données sont d'abord effectuées séquentiellement dans un WAL, nommé le *commit log*. Elles sont ensuite envoyées aux nœuds en fonction du type d'écriture, comme nous le verrons dans la section suivante consacrée à la mise en œuvre de Cassandra. Chaque nœud écrit la donnée dans son commit log et l'ajoute en mémoire dans une *memtable*. La memtable est l'équivalent d'un buffer, c'est une représentation en mémoire de paires clé-valeur de données d'une famille de colonnes. Chaque famille de colonnes a sa memtable. Elle est écrite sur le disque après un certain délai, par exemple si la mémoire est saturée ou si un nombre défini de clés a été ajouté. Le fichier enregistré sur le disque s'appelle une SSTable *(Sorted Strings Table)*. Un filtre de Bloom est aussi maintenu, qui est une structure permettant de déterminer si une clé est présente ou non dans la SSTable en utilisant un algorithme basé sur les probabilités. C'est une façon élégante d'améliorer les performances d'accès aux données. Chaque fois qu'une memtable est sauvegardée, elle s'écrit dans une nouvelle SSTable. Il peut donc y avoir plusieurs SSTables pour la même famille de colonnes et les SSTables sont immutables : il y aura toujours création de nouvelles SSTables, jamais de modification d'un SSTable existante. Régulièrement, un compactage crée une nouvelle SSTable à partir des existantes.

Mise en œuvre

Installation

Vous pouvez obtenir Cassandra de deux manières : depuis Apache, ou depuis les dépôts de Datastax, une société qui s'est développée autour de ce moteur. Datastax dispose d'une version community que nous allons utiliser ici.

Un dépôt pour Debian (donc compatible avec Ubuntu) est disponible en ligne. Vous trouverez les explications nécessaires à l'adresse suivante : http://www.datastax.com/documentation/cassandra/2.0/cassandra/install/installDeb_t.html.

Nous ajoutons les dépôts à notre fichier `/etc/apt/sources.list` :

```
deb http://debian.datastax.com/community stable main
```

Avant l'installation, nous importons la clé publique `gpg` (*GNU Privacy Guard*, l'équivalent libre de PGP) qui permettra de vérifier la signature du paquet afin de garantir son authenticité :

```
curl -L http://debian.datastax.com/debian/repo_key | sudo apt-key add -
```

Ensuite, nous installons Cassandra comme tout autre paquet :

```
sudo apt-get update
sudo apt-get install dsc20=2.0.11-1 cassandra=2.0.11
```

Comme vous le voyez, nous indiquons la version, ce qui peut être utile pour éviter ensuite l'écrasement automatique par une nouvelle version lors d'un upgrade avec `apt-get` (dsc signifie DataStax Community).

L'installation du paquet démarre le démon Cassandra. Nous pouvons nous en assurer en le cherchant dans les processus du système, ou en vérifiant s'il écoute sur son port TCP par défaut, le port 9160.

```
netstat -lt | grep 9160
```

Afin de permettre l'accès au serveur depuis une machine distante, nous changeons la configuration de Cassandra située dans le fichier `/etc/cassandra/cassandra.yaml`, en modifiant la ligne suivante :

```
rpc_address: localhost
```

en :

```
rpc_address: 0.0.0.0
```

Puis nous redémarrons :

```
sudo service cassandra restart
```

Configuration

Le fichier de configuration de Cassandra est situé sur notre système à `/etc/cassandra/cassandra.yaml`. Le tableau 11-1 liste quelques éléments pour parcourir les possibilités de Cassandra. Nous avons déjà vu le paramètre `snitch` dans le chapitre 3 dédié à la distribution des données.

Tableau 11-1. Éléments de configuration

Option de configuration	Description
`cluster_name`	Nom logique du cluster, permet d'identifier les nœuds faisant partie d'un même cluster.
`initial_token`	La valeur de token pour ce nœud dans l'anneau, c'est-à-dire une valeur numérique de `crc32` qui sera la rangée de ce nœud dans l'anneau (voir la section sur le hachage consistant du chapitre 3, page 78). Cette option est dépréciée et ne doit être utilisée que sur des machines qui vont joindre un cluster existant. Pour les nouveaux clusters, voyez l'option suivante.
`num_tokens`	Le nombre de vnodes à héberger sur ce nœud. Le défaut est 1 pour désactiver la fonctionnalité de vnodes par souci de compatibilité. Mettez 256 pour utiliser les vnodes, c'est la valeur recommandée.
`Authenticator`	Permet d'indiquer une classe Java pour authentifier les connexions au serveur.
`Authority`	Permet d'indiquer une classe Java pour implémenter les permissions.
`key_cache_size_in_mb`	Taille du cache de clés. Une zone de mémoire dédiée à la conservation des clés. Très intéressant pour améliorer les performances. Le stockage des seules clés en mémoire est compact. Le cache peut être sauvegardé régulièrement sur le disque pour être rechargé au démarrage de Cassandra.
`row_cache_size_in_mb`	Taille du cache de lignes. Moins intéressant que le cache de clés, car sa taille en mémoire est plus importante, et sa maintenance est plus lourde si les valeurs sont régulièrement mises à jour.

L'API Thrift

Historiquement, l'accès à Cassandra était réalisé via Thrift. L'outil en ligne de commande `cassandra-cli` était – est toujours car il existe pour compatibilité – un moyen interactif pour nous connecter au serveur :

```
cassandra-cli --host localhost --port 9160
```

Et c'est l'outil que nous avions montré dans la première édition de ce livre. Depuis, la révolution Cassandra 2 est passée. Cassandra 2 a déprécié l'API Thrift au profit de CQL. CQL est un langage déclaratif semblable à SQL, qui utilise un protocole binaire de type RPC pour communiquer avec le serveur. De ce fait, les performances sont meilleures qu'avec l'API Thrift (voyez par exemple ceci http://www.datastax.com/dev/blog/cassandra-2-1-now-over-50-faster), tout en offrant un langage facile à manipuler et familier à tout développeur ayant déjà fait du SQL, c'est-à-dire tout développeur…

Juste pour savoir de quoi nous parlons, voici un exemple basique d'ajout de données à l'aide de commandes de l'API Thrift :

```
assume articles keys as utf8;
set articles[1]['auteur_prenom'] = 'Annie';
set articles[1]['auteur_nom'] = 'Brizard';
set articles[1]['auteur_email'] = 'annie.brizard@cocomail.com';
set articles[1]['titre'] = 'pourquoi les éléphants ont-ils de grandes oreilles?';
```

Vous voyez le genre. Passons donc à CQL, qui l'a remplacé.

CQL 2 et 3

Cassandra implémente donc un langage de requête proche du SQL, nommé CQL *(Cassandra Query Language)*. Vous le voyez dans le titre de cette section, il y a eu deux versions importantes de CQL. Pourquoi parler d'une ancienne version ? pour illuster un changement important.

CQL 2 a été défini comme un ajout à l'API Thrift. Il offrait moins de fonctionnalités et se basait conceptuellement sur Thrift. Par exemple, vous pouviez en CQL 2 faire ceci :

```
CREATE COLUMNFAMILY articles (
        KEY bigint PRIMARY KEY,
        "auteur:prenom" varchar,
        «auteur:nom» varchar
);
```

Puis ceci :

```
INSERT INTO articles (KEY, «auteur:prenom», «auteur:nom», «auteur:email»)
VALUES (1, 'Frank', 'Lavue', 'frank.lavue@cocomail.com');
```

Vous voyez ? Nous insérons le contenu d'une colonne, `auteur:email`, que nous n'avons pas défini dans le `CREATE COLUMNFAMILY`. CQL 2 n'est qu'une surcouche.

Cela n'est plus possible en CQL 3. Pour travailler avec une famille de colonnes, elle doit avoir avoir été créée, avec le bon nombre de colonnes et même le bon type de données. Au lieu d'être juste un sucre de syntaxe, CQL 3 est une couche d'abstraction qui cache la structure interne du stockage Cassandra, à tel point que… voyons un exemple différent :

```
CREATE TABLE commentaires (
        article_id bigint,
```

```
    date_commentaire timestamp,
       auteur varchar,
       contenu text,
    PRIMARY KEY (article_id, date_commentaire)
);
```

Ici, nous voulons stocker des commentaires postés sur un article de blog, référencé par un `article_id`. Quelles sont les différences avec la déclaration précédente ? D'abord, nous disons `CREATE TABLE` au lieu de `CREATE COLUMNFAMILY`. Ce sont des synonymes mais on se rapproche du ressenti d'un SGBDR. Ensuite, nous indiquons un nom pour chaque colonne, là où nous avions indiqué auparavant `KEY` pour la row key. Ensuite, nous définissons une clé primaire composite ! On l'a vu, Cassandra stocke chaque ligne par rapport à une row key. Cette row key peut-elle être composite ? Non, et ce que nous venons de déclarer en CQL 3 sera stocké de façon toute particulière dans Cassandra, sous forme de ce qui est appelé une *wide row*. En réalité, chaque commentaire sur le même article sera stocké dans la même ligne, et la row key de cette ligne sera l'`article_id`. Voici un exemple de stockage interne de deux commentaires sur le même article (l'article 1) postés le 1er janvier 2015 et le 10 janvier 2015. Nous représentons ici le contenu d'une seule ligne dans Cassandra, en montrant les types de données de façon logique, et non pas tels qu'ils sont physiquement stockés :

```
Clé   COLONNE              VALEUR
1:    2015-01-01:
      2015-01-01:auteur    Jean Posteur
      2015-01-01:contenu   Tout à fait d'accord
      2015-01-10:
      2015-01-10:auteur    Anne Lafleur
      2015-01-10:contenu   Bon article
```

En fait, chaque insertion dans cette table sur le même `article_id` va générer une création, non pas d'une nouvelle ligne dans la table, mais de trois nouvelles colonnes sur la ligne identifiée par `article_id`. Cela va permettre de conserver tous les commentaires sur le même article sur le même nœud Cassandra pour un accès plus rapide. Cassandra compacte ces données, donc l'impact en stockage sera peu important, et CQL 3 compose et décompose la ligne lorsqu'il la manipule.

Gestion de la cohérence

Dans Cassandra, la cohérence des écritures et des lectures est configurable par instruction. Les options de cohérence sont résumées dans le tableau 11-2.

Tableau 11-2. **Modes de respect de la cohérence**

Niveau	Description
ANY	L'écriture doit être faite sur un nœud au moins, même si c'est un hinted handoff. Disponible en écriture uniquement.
ONE	L'écriture ou la lecture doit être réalisée sur un nœud au moins. L'écriture doit être inscrite dans le commit log et la memtable.
QUORUM	L'écriture ou la lecture doit être au moins effectuée sur le quorum des réplicas de cette donnée, c'est-à-dire la moitié des réplicas plus un. L'écriture doit être inscrite dans le commit log et la memtable de ces réplicas.
LOCAL_QUORUM	L'écriture ou la lecture doit être faite au moins sur le quorum des réplicas de cette donnée, présents dans le data center local. Cela permet de s'affranchir des délais dus à la présence de réplicas sur d'autres data centers. L'écriture doit être inscrite dans le commit log et la memtable des réplicas.
EACH_QUORUM	L'écriture ou la lecture doit être au moins réalisée sur un quorum des réplicas de cette donnée, présents dans chaque data center.
ALL	L'écriture ou la lecture doit être effectuée sur tous les réplicas.

Comme vous le voyez, nous ne sommes pas ici dans une cohérence finale obligatoire, mais nous pouvons décider au cas par cas. Le niveau par défaut est `ONE`. Voici comment modifier le niveau de cohérence pour la session en CQL :

```
CONSISTENCY QUORUM;
```

Au niveau des lectures, Cassandra dispose d'une fonctionnalité de *Read Repair* automatique. Une lecture directe entraîne une lecture asynchrone en tâche de fond sur les réplicas qui ne sont pas affectés par la demande de lecture, afin de détecter éventuellement des changements et de les répercuter sur les autres réplicas.

Hinted handoff

La fonctionnalité hinted handoff permet de synchroniser un nœud qui est indisponible pour une petite période de temps. Une écriture est toujours envoyée à tous les réplicas. Si un réplica n'est pas disponible, les autres réplicas vont garder une information indiquant qu'une écriture est en attente pour ce nœud, et lorsqu'il est de nouveau disponible, ils lanceront l'écriture.

Programmation client

Dans l'édition précédente de cet ouvrage, nous avions utilisé le client Python pycassa (https://github.com/pycassa/pycassa). Cette bibliothèque utilise le protocole Thrift. Si vous êtes en Cassandra 2, elle est dépréciée. Datastax, la société qui distribue et concourt au développement de Cassandra, a conçu un client Python qui utilise le protocole CQL (https://github.com/datastax/python-driver). Nous allons donc l'utiliser dorénavant. Si vous souhaitez travailler plutôt en objets qu'en CQL, un ORM (outil de mapping relationnel-objet) pour Python nommé cqlengine (https://github.com/cqlengine/cqlengine) offre une couche d'abstraction supplémentaire. Nous allons vous montrer comment utiliser les deux.

D'abord, pour compiler au mieux le pilote, nous installons des bibliothèques sur Ubuntu :

```
sudo apt-get install libev4 libev-dev
```

Elles vont permettre la compilation de cassandra.io.libevwrapper, laquelle est l'extension qui va travailler directement avec l'algorithme de hachage utilisé par Cassandra et permettre de diriger les appels vers le bon nœud. Sans cela, le client fonctionnera mais sera plus lent, car il appellera un nœud qui redirigera la requête, au lieu d'aller directement au bon endroit.

Puis nous installons le pilote et sqlengine :

```
~/python_env/nosql/bin/pip install cassandra-driver
```

Nous allons utiliser notre traditionnel exemple Passerelles et ajouter un article, puis le récupérer.

```
#! /usr/bin/python
# -*- coding: utf8 -*-

import uuid
import datetime
from cqlengine import columns
from cqlengine.models import Model

class Article(Model):
    read_repair_chance = 0.05 # optional - defaults to 0.1
    article_id      = columns.UUID(primary_key=True, default=uuid.uuid4)
    auteur_prenom   = columns.Text(required=True)
    auteur_nom      = columns.Text(required=True, index=True)
    date            = columns.DateTime()
    titre           = columns.Text(required=True)
    article         = columns.Text(required=True)

from cqlengine import connection
connection.setup(['192.168.0.16'], "passerelles")

from cqlengine.management import create_keyspace
create_keyspace("passerelles", "SimpleStrategy", 1)

from cqlengine.management import sync_table
sync_table(Article)

ar = Article.create(auteur_prenom=»Annie», auteur_nom=»Dakota», date=datetime.
datetime.now(), titre=»Les éléphants ont-ils de grande oreilles?», article=»une longue
histoire» )

Article.objects.count()

q = Article.objects(auteur_nom = «Dakota»)
for instance in q:
    print instance.titre
```

Nous créons d'abord une classe dont les propriétés correspondent aux types Cassandra. Nous ouvrons ensuite une connexion avec cqlengine.

Puis nous utilisons la méthode `create_keyspace` pour créer un keyspace, donc une base de données. Nous utilisons alors la méthode `sync_table` pour créer la table, et nous insérons une ligne.

Nous effectuons finalement une recherche sur une colonne indexée avec la méthode `Article.objects()`.

Nodetool

L'utilitaire `nodetool` permet d'effectuer des opérations de maintenance ou d'obtenir des informations sur le nœud Cassandra.

Par exemple, la commande :

```
nodetool ring
```

retourne la liste des machines dans l'anneau. Pour notre exemple, il n'y en a qu'une, voici le résultat :

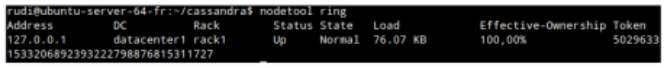


Figure 11-1
Résultat de nodetool ring

Nous voyons les informations de data center (DC), de rack et le token, telles que nous les avons configurées.

La commande :

```
nodetool cfstats
```

retourne des statistiques sur les familles de colonnes. Voici un extrait du résultat pour la famille de colonnes `passerelles` :

```
Keyspace: passerelles
        Read Count: 9
        Read Latency: 4.22 ms.
        Write Count: 11
        Write Latency: 0.3235454545454546 ms.
        Pending Tasks: 0
                Column Family: articles
                SSTable count: 1
                Space used (live): 4798
                Space used (total): 4798
                Number of Keys (estimate): 128
                Memtable Columns Count: 10
                Memtable Data Size: 275
                Memtable Switch Count: 1
                Read Count: 9
                Read Latency: 4,220 ms.
                Write Count: 11
                Write Latency: 0,324 ms.
```

```
                Pending Tasks: 0
                Bloom Filter False Postives: 0
                Bloom Filter False Ratio: 0,00000
                Bloom Filter Space Used: 24
                Compacted row minimum size: 87
                Compacted row maximum size: 258
                Compacted row mean size: 192
```

Ces informations sont précieuses pour juger des performances de Cassandra.

L'appel suivant retire un nœud de l'anneau :

```
nodetool disablegossip
```

Pour le faire revivre, nous lançons ensuite la commande suivante :

```
nodetool enablegossip
```

Puis nous vérifions que le nœud communique avec les autres :

```
nodetool netstats
```

Conclusion

Cassandra est un des moteurs les plus complets et les mieux bâtis, il vous permet d'allier les avantages d'un moteur orienté colonnes avec l'architecture décentralisée et les capacités de montée en charge automatique de Dynamo. C'est un choix intéressant pour les environnements critiques.

12

Les autres bases de données de la mouvance NoSQL

Nous vous avons présenté les moteurs de bases de données NoSQL les plus importants. Il en existe beaucoup d'autres, s'inspirant plus ou moins des mêmes techniques, mais ils sont pour la plupart moins aboutis et moins utilisés que ceux que nous avons vus.

Dans ce chapitre, nous souhaitons vous présenter deux types d'outils très intéressants qu'il est également possible de classer dans la catégorie des moteurs NoSQL : le moteur de recherche ElasticSearch et le moteur orienté graphes, Neo4j.

ElasticSearch

ElasticSearch n'est pas un système de gestion de bases de données. Il s'agit d'un moteur de recherche plein texte à indexation automatique, développé par Shay Banon et basé sur Lucene. Il est principalement utilisé pour alimenter la recherche plein texte sur des sites web. Cet outil est donc semblable au moteur de recherche de Google, par exemple.

> **Lucene**
> Lucene est un célèbre moteur de recherche plein texte de la fondation Apache. C'est une application Java développée par Doug Cutting, lequel créera plus tard Hadoop, et qui offre d'excellentes performances.

ElasticSearch constitue une surcouche de Lucene, dont il reprend les fonctionnalités de recherche. Il permet de répartir le travail sur plusieurs machines et offre également une interface REST

afin d'accéder très simplement à ses fonctionnalités. Grâce à cette interface REST, il est possible d'envoyer des documents JSON à indexer et d'effectuer des recherches. Il s'agit donc d'un moteur de recherche orienté documents, ce qui le rapproche beaucoup du mouvement NoSQL. Passons maintenant à la pratique.

Pour commencer, rendez-vous sur le site d'ElasticSearch pour télécharger le moteur de recherche : http://www.elasticsearch.org/download/.

En cliquant sur le numéro de version que vous souhaitez, vous y trouverez un paquet Debian (extension `.deb`), que vous devez récupérer et installer. Nous avons pour notre part récupéré le lien du paquet et nous l'avons téléchargé avec `wget` :

```
wget https://github.com/downloads/elasticsearch/elasticsearch/elasticsearch-1.4.4.deb
```

Puis nous installons le paquet avec `dpkg`. À la fin de l'installation, le démon est automatiquement démarré.

```
sudo dpkg -i elasticsearch-1.4.4.deb
```

ElasticSearch peut ensuite être arrêté et lancé comme un service :

```
sudo service elasticsearch stop
sudo service elasticsearch start
```

Sur notre système, le fichier de configuration est `/etc/elasticsearch/elasticsearch.yml`. Nous pouvons y paramétrer les informations du cluster, comme son nom ainsi que les attributs du nœud qui pourront être utilisés comme paramètres pour le sharding des données d'indexation. La topologie est autant que possible autoconfigurée, mais nous pouvons intervenir manuellement dans ce fichier pour déterminer quelles sont les responsabilités du nœud, entre nœud maître et nœud de données.

Mise en œuvre

ElasticSearch répartit ses index de la même manière que les moteurs NoSQL distribués que nous avons étudiés précédemment, c'est-à-dire en effectuant un sharding et une réplication des données. Pour son interface REST, ElasticSearch écoute par défaut sur toutes les interfaces, et sur le port 9200. Si nous appelons l'interface REST à cette adresse, nous obtenons :

```
{
  "status" : 200,
  "name" : "Mentus",
  "version" : {
    "number" : "1.2.0",
    "build_hash" : "c82387f290c21505f781c695f365d0ef4098b272",
    "build_timestamp" : "2014-05-22T12:49:13Z",
    "build_snapshot" : false,
    "lucene_version" : "4.8"
  },
  "tagline" : "You Know, for Search"
}
```

Grâce à un appel REST, nous allons directement créer un index nommé `passerelles`. On peut considérer ici que l'index équivaut plus ou moins à une base de données :

```
curl -X PUT http://localhost:9200/passerelles
```

Nous recevons cette notification :

```
{«ok»:true,»acknowledged»:true}
```

Nous stockons ensuite un document :

```
curl -X PUT http://localhost:9200/passerelles/articles/1 -d '
{
        «auteur»:
        {
            «prénom»:»Annie»,
            «nom»:»Brizard»,
            «e-mail»:»annie.brizard@cocomail.com»
        },
        «titre»:»pourquoi les éléphants ont-ils de grandes oreilles?»
}'
```

Mapping des types

Chaque élément du document est mappé à un type de donnée. ElasticSearch reconnaît les types de données JSON de façon à mieux qualifier la donnée dans la recherche. Ce mapping peut aussi être effectué manuellement, à l'aide d'une API de mapping. Les types sont représentés par des classes de `Mapper`, qui s'enrichissent au fil du temps. Par exemple, il existe des classes de mapping pour les points et les formes géospatiales, qui utilisent des bibliothèques géospatiales pour Java. Le mapping permet de déterminer le comportement de l'indexation par rapport aux différents champs d'un document.

Nous avons créé le document `1` dans l'index `passerelles`. Comme vous l'avez sans doute remarqué, il existe un niveau intermédiaire appelé `type`, dont la valeur est `articles` pour notre exemple. Le type est créé automatiquement lorsque vous ajoutez un document, il équivaut plus ou moins à une table. Nous obtenons la confirmation suivante :

```
{«ok»:true,»_index»:»passerelles»,»_type»:»articles»,»_id»:»1»,»_version»:1}
```

On voit ici qu'un numéro de version est également attribué au document, tout ceci ressemble très fortement à un moteur NoSQL. D'ailleurs, comme on peut le faire dans un moteur tel que CouchDB, il nous suffit d'utiliser un `GET` pour retrouver notre document :

```
curl -i http://localhost:9200/passerelles/articles/1
```

et nous obtenons :

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 259
```

```
{«_index»:»passerelles»,»_type»:»articles»,»_id»:»1»,»_version»:1,»exists»:true, «_
source»:
{
«auteur»:
{
«prénom»:»Annie»,
«nom»:»Brizard»,
«e-mail»:»annie.brizard@cocomail.com»
},
«titre»:»pourquoi les éléphants ont-ils de grandes oreilles?»
}}
```

La présence d'un numéro de version est utile pour gérer un verrouillage optimiste, comme dans CouchDB.

Dans la mesure où il s'agit d'un moteur d'indexation, effectuons donc une recherche. Le plus simple est d'envoyer une chaîne de recherche dans l'URL. Dans l'exemple suivant, nous voulons trouver les articles dont le titre contient le mot «oreilles»:

```
curl -i 'http://localhost:9200/passerelles/_search?q=titre:oreilles'
```

Nous obtenons:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 384

{"took":175,"timed_out":false,"_shards":{"total":5,"successful":5,"failed":0},"hits":{
«total»:1,»max_score»:0.095891505,»hits»:[{«_index»:»passerelles»,»_
type»:»articles»,»_id»:»1»,»_score»:0.095891505, «_source»:
{
«auteur»:
{
«prénom»:»Annie»,
«nom»:»Brizard»,
«e-mail»:»annie.brizard@cocomail.com»
},
«titre»:»pourquoi les éléphants ont-ils de grandes oreilles?»
}]}}
```

Si les données sont shardées, ElasticSearch lancera la recherche sur tous les shards et renverra le résultat regroupé.

Plug-ins et rivières

Les fonctionnalités d'ElasticSearch peuvent être enrichies à l'aide de plug-ins, et notamment ce que l'on appelle des «rivières». Ces dernières sont des connecteurs à des sources de données qui permettent de déverser au fil de l'eau les nouvelles données à indexer dans ElasticSearch. Des rivières existent pour différents types de sources, bases de données NoSQL ou SQL, sites

comme Wikipédia ou réseaux sociaux tels que Twitter, etc. Elles sont développées par l'équipe d'ElasticSearch ou par des développeurs tiers. Des rivières existent notamment pour MongoDB et CouchDB. Dans ce dernier, la fonctionnalité de suivi des changements, dont nous n'avons pas encore parlé, permet d'envoyer les données vers ElasticSearch. Voyons tout cela à travers un exemple.

Rivière CouchDB

CouchDB offre une fonctionnalité de suivi de changements à travers l'API `_changes`. Au plus simple, en appelant `_changes` sur une base de données, nous obtenons la liste des modifications qui y ont été apportées. Essayons sur notre base `passerelles` :

```
curl http://nosql-server:5984/passerelles/_changes
```

Nous obtenons ceci :

```
{«results»:[
{«seq»:2,»id»:»id_article_1»,»changes»:[{«rev»:»2-f6c92fbe3b7579f0f6819e1c85db3eaa»}],
“deleted”:true},
{“seq”:4,”id”:”1”,”changes”:[{“rev”:”2-1a97e3eb387fd93cf6e1abb2a1aed56b”}]},
{“seq”:6,”id”:”a642aa5b272c49d72aab7aef26226508”,”changes”:[{“rev”:”2-5c8ec9a6b2973061
2c2eba48ea845943”}],”deleted”:true},
{“seq”:7,”id”:”a642aa5b272c49d72aab7aef26415ce4”,”changes”:[{“rev”:”1-a0e81b6cc21cb1e5
52dad87ec4c41f49”}]},
{“seq”:8,”id”:”a642aa5b272c49d72aab7aef266097d5”,”changes”:[{“rev”:”1-a0e81b6cc21cb1e5
52dad87ec4c41f49”}]},
{“seq”:9,”id”:”a642aa5b272c49d72aab7aef267e7951”,”changes”:[{“rev”:”1-a0e81b6cc21cb1e5
52dad87ec4c41f49”}]},
{“seq”:12,”id”:”2”,”changes”:[{“rev”:”3-7379b9e515b161226c6559d90c4dc49f”}],”deleted”:
true},
{“seq”:13,”id”:”_design/
python”,”changes”:[{“rev”:”1-c40cae63adc999b7ed184e6324da50bd”}]}
],
«last_seq»:13}
```

S'affiche donc la liste des modifications qui contient les clés, les identifiants de versions et les types de changements. Nous obtenons aussi des numéros de séquences, qui représentent l'ordre des changements, ainsi que le dernier numéro de séquence (`13`) pour terminer. Dans notre application, il suffit de stocker ce dernier numéro et de l'indiquer de la manière suivante au prochain appel :

```
curl http://nosql-server:5984/passerelles/_changes?since=13
```

pour récupérer les modifications effectuées après le numéro de séquence `13`, et ainsi de suite.

Le suivi des changements peut aussi être appelé en *long polling*, c'est-à-dire en maintenant une connexion HTTP jusqu'à ce qu'un résultat soit obtenu. La commande suivante :

```
curl http://nosql-server:5984/passerelles/_changes?feed=longpoll&since=13
```

ne rendra la main que lorsqu'une modification sera apportée à la base `passerelles`. Enfin, le mode continu laisse la connexion ouverte et récupère au fil de l'eau les modifications en temps réel :

```
curl http://localhost:5984/passerelles/_changes?feed=continuous&since=13
```

Configuration et utilisation de la rivière

Comme mentionné précédemment, une rivière est un plug-in d'ElasticSearch. Il convient donc de l'installer à l'aide de la commande `plugin` située dans le répertoire `bin` d'ElasticSearch :

```
/usr/share/elasticsearch/bin/plugin -install elasticsearch/
elasticsearch-river-couchdb/1.1.0
```

Ensuite, pour configurer la rivière, nous exécutons simplement la commande REST suivante, qui envoie les informations de configuration dans son JSON.

```
curl -XPUT 'localhost:9200/_river/passerelles/_meta' -d '{
    «type»: «couchdb»,
    «couchdb»: {
        «host»: «localhost»,
        «port»: 5984,
        «db»: «passerelles»,
        «filter»: null
    },
    «index»: {
        «index»: «passerelles»,
        «type»: «passerelles»,
        «bulk_size»: «100»,
        «bulk_timeout»: «10ms»
    }
}'
```

Nous indiquons ici les informations permettant à ElasticSearch de se connecter à CouchDB et de récupérer les données à travers le suivi de changement, pour les indexer en temps réel dans son index `passerelles`.

Journal d'ElasticSearch

Si vous rencontrez des problèmes, consultez le log d'ElasticSearch afin d'obtenir des informations détaillées. Son emplacement est défini dans le fichier de configuration `logging.yml` (dans `/etc/elasticsearch` pour notre configuration). Sur notre installation Ubuntu, il se trouve dans `/var/log/elasticsearch`.

Pour effectuer un test, nous ajoutons ensuite un document dans CouchDB :

```
curl -X PUT http://localhost:5984/passerelles/2 -d '
{
        «auteur»:
        {
            «prénom»:»Jean»,
            «nom»:»Crou»,
            «e-mail»:»jcrou@cocomail.com»
        },
```

```
        «titre»:»chroniques du quotidien»
}'
```

Puis nous effectuons la recherche dans ElasticSearch :

```
curl -i 'http://localhost:9200/passerelles/_search?q=titre:quotidien'
```

Nous obtenons:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 365

{"took":51,"timed_out":false,"_shards":{"total":5,"successful":5,"failed":0},"hits":{"
total»:1,»max_score»:0.5,»hits»:[{«_index»:»passerelles»,»_type»:»passerelles»,»_
id":"2","_score":0.5, "_source": {"_rev":"1-bb2ef2e7d8366889c36408ae44bb0a91","_
id»:»2»,»titre»:»chroniques du quotidien»,»auteur»:{«prénom»:»Jean»,»e-mail»:»jcrou@
cocomail.com»,»nom»:»Crou»}}]}}
```

Les facettes

Les facettes représentent l'une des fonctionnalités majeures d'ElasticSearch. En plus de retourner un résultat de recherche, ce dernier peut également retourner des informations agrégées sur ces résultats, par exemple un compte d'occurrences, un histogramme de dates, etc. Les facettes disponibles sont documentées à l'adresse suivante: http://www.elasticsearch.org/guide/reference/api/search/facets/index.html. Reprenons notre exemple d'index `passerelles` qui intègre la base `passerelles` de CouchDB. Nous allons analyser les e-mails pour déterminer la fréquence d'apparition d'utilisateurs et de domaines, à l'aide de la requête suivante:

```
curl -X POST "http://localhost:9200/passerelles/_search?pretty=true" -d '
{
    "query": {
        "match_all": { }
    },
    "facets": {
        "e-mails": {
            «terms»: {
                «field»: «e-mail»
            }
        }
    }
}
```

Nous demandons à ElasticSearch de nous retourner toutes les occurrences d'articles, et d'ajouter un calcul de facette que nous nommons `e-mails`, en utilisant la facette `terms`. Le résultat fragmentaire est le suivant (nous n'avons conservé que la partie facette):

```
  "facets": {
    "e-mails": {
      "_type": "terms",
```

```
      "missing": 1,
      "total": 4,
      "other": 0,
      "terms": [ {
        "term": "cocomail.com",
        "count": 3
      }, {
        "term": "annie.brizard",
        "count": 2
      }, {
        «term»: «jcrou»,
        «count»: 1
      } ]
    }
  }
```

Nous voyons qu'un document ne contient pas de champ e-mail (il est indiqué par `missing`). Il y a donc trois articles qui sont analysés. On trouve trois fois `cocomail.com`, deux fois `annie.brizard` et une fois `jcrou`.

Les bases de données orientées graphes

Le concept des bases de données orientées graphes existait déjà avant l'invention du modèle relationnel. Les implémentations modernes sont performantes et intéressantes pour certains cas d'utilisation, qui correspondent à des parcours d'arbres à la recherche de relations significatives à travers les différents niveaux. Par exemple, découvrir les amis des amis d'un utilisateur qui habitent dans la même ville, calculer un trajet d'un point vers un autre selon des critères de temps ou de distance, représenter la traçabilité de produits ou de flux financiers, etc.

Neo4j

Neo4j est l'une des meilleures bases de données orientée graphes. Bien qu'elle soit développée en Java, elle offre d'excellentes performances et permet de traiter rapidement de grandes quantités de relations.

Nous allons installer Neo4j sur notre système Ubuntu. Les paquets Debian/Ubuntu sont disponibles à l'adresse suivante : http://debian.neo4j.org/. Il suffit d'ajouter le dépôt dans la liste de nos sources de paquets, en créant par exemple un fichier `/etc/apt/sources.list.d/neo4j.list` qui contient cette ligne :

```
deb http://debian.neo4j.org/repo testing/
```

Nous voulons utiliser la dernière version de Neo4j, soit la 1.8 à l'heure où nous écrivons ces lignes. Si vous installez Neo4j en situation de production, choisissez le paquet `stable`, en rédigeant vos sources comme suit :

```
deb http://debian.neo4j.org/repo stable/
```

Nous rafraîchissons ensuite la liste des paquets et nous pouvons installer l'une des trois livraisons :

```
sudo apt-get update && sudo apt-get install neo4j-advanced
```

Neo4j s'installe et démarre en tant que service. Nous pouvons maintenant accéder au serveur web d'administration : http://localhost:7474/webadmin/. Si vous souhaitez pouvoir accéder à la page d'administration depuis une machine distante, modifiez le fichier `/etc/neo4j/neo4j-server.properties` et décommentez cette ligne :

```
org.neo4j.server.webserver.address=0.0.0.0
```

Redémarrez ensuite le service :

```
sudo service neo4j-service restart
```

L'interface d'administration est assez plaisante. La figure 12-1 représente l'écran d'accueil.

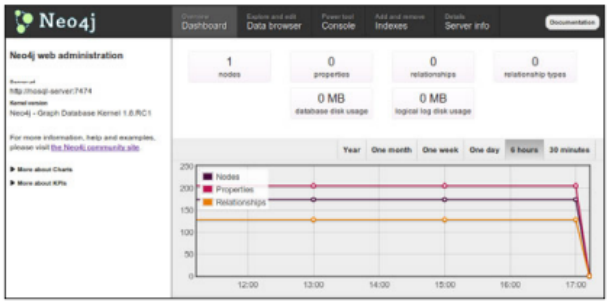


Figure 12-1
L'interface d'administration de Neo4j

Mise en œuvre

La figure 12-1 montre une instance de Neo4j où nous venons de supprimer des données en vidant la base de données. Le graphique indique le nombre de nœuds, qui vient donc subitement de chuter, à cause de notre suppression.

> **La base de données Neo4j**
>
> Neo4j ne gère pas les bases de données multiples. Le cas d'utilisation d'une base de données orientée graphes est spécifique, et il n'y a pas vraiment de sens de créer plusieurs bases de données. Neo4j stocke toutes ses données dans un répertoire. Pour supprimer la base de données entière, il convient d'arrêter le service, de supprimer tous les fichiers du répertoire (`/var/lib/neo4j/data/graph.db/` pour notre configuration) et de redémarrer le service.

Nous basculons sur l'onglet Console et nous allons saisir quelques nœuds en utilisant le langage Cypher.

Les langages de Neo4j

Mis à part les pilotes spécifiques pour les langages clients, qui permettent de manipuler les nœuds et les relations dans un mode procédural, Neo4j offre deux langages spécifiques : Cypher et Gremlin. Cypher est un langage déclaratif inspiré du SQL, qui permet d'exprimer une requête complexe de façon élégante et compacte. Gremlin, quant à lui, est un langage de script basé sur Groovy (un langage pour la plate-forme Java dont la syntaxe s'inspire de langages de script comme Python et Ruby). Il permet d'envoyer des scripts qui seront exécutés du côté serveur à travers l'interface REST de Neo4j. Nous allons utiliser ici des commandes Cypher. Ces dernières peuvent également être lancées en ligne de commande avec l'invite `neo4j-shell`.

Tout d'abord, nous créons un nœud :

```
CREATE n = {nom: 'Milou', ville: 'Paris'};
```

Le premier nœud porte le numéro 1, comme nous pouvons le vérifier dans l'interface d'administration de Neo4j, onglet Data browser, en saisissant simplement « 1 » dans le champ de saisie et en cliquant sur la loupe. Le résultat est affiché sur la copie d'écran de la figure 12-2.

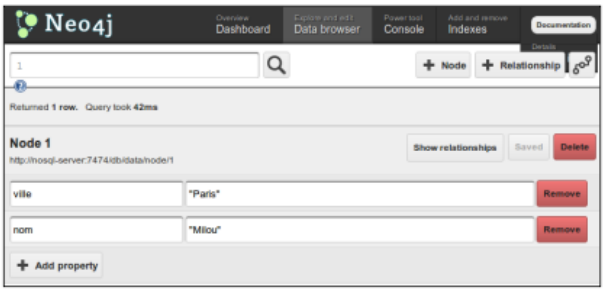


Figure 12-2
L'onglet Data browser de l'interface d'administration de Neo4j

Nous pouvons bien sûr obtenir la même chose en Cypher :

```
START n=node(1) RETURN n;
```

qui retourne :

```
==> +-----------------------------------+
==> | n                                 |
==> +-----------------------------------+
==> | Node[1]{nom:»Milou»,ville:»Paris»} |
==> +-----------------------------------+
==> 1 row
==> 0 ms
```

Nous allons maintenant ajouter un deuxième nœud et une relation dans la même commande :

```
START Milou = node(1)
```

```
CREATE
    Tintin = { nom: 'Tintin', ville: 'Paris' },
    Milou-[r:AMI]->Tintin
RETURN
    Tintin;
```

ce qui retourne :

```
==> +-----------------------------------+
==> | Tintin                            |
==> +-----------------------------------+
==> | Node[2]{nom:»Tintin»,ville:»Paris»} |
==> +-----------------------------------+
==> 1 row
==> Nodes created: 1
==> Relationships created: 1
==> Properties set: 2
==> 42 ms
==>
```

À l'aide de la commande `START`, nous avons indiqué le point de départ de notre pattern, c'est-à-dire de notre instruction. Nous avons ensuite créé un nœud et une relation que nous avons nommée `AMI`. Ajoutons encore quelques nœuds :

```
START Milou = node(1), Tintin = node(2)
CREATE
    Haddock = { nom: 'Capitain Haddock', ville: 'Moulinsart' },
    Castafiore = { nom: 'Castafiore', ville: 'Pesaro' },
    Tintin-[r:AMI]->Haddock,
     Haddock -[r:AMI]->Castafiore;
```

Nous retournons maintenant dans l'onglet Data browser et nous activons l'affichage graphique des relations (figure 12-3).

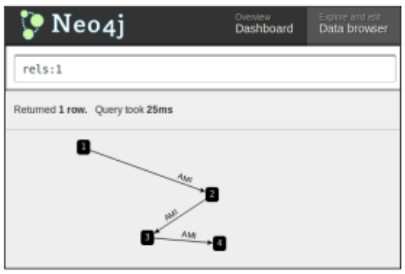


Figure 12-3
Affichage graphique des relations

Les relations, comme les nœuds, sont numérotées. Nous pouvons maintenant appliquer toutes sortes de requêtes pour traverser l'arbre et chercher les relations. Par exemple, la requête suivante cherche s'il y a une relation de type `AMI` entre Milou et la Castafiore :

```
START Milou=node(1), Castafiore=node(4)
MATCH Milou-[r:AMI*]->Castafiore
RETURN r;
```

La requête peut se lire ainsi : « pour Milou et la Castafiore, s'il y a une relation de type `AMI` qui va de Milou à la Castafiore en passant par un nombre indéfini (*) de relations intermédiaires, retourne-moi cette relation ». Le résultat est :

```
==> +-----------------------------------+
==> | r                                 |
==> +-----------------------------------+
==> | [:AMI[2] {},:AMI[1] {},:AMI[0] {}] |
==> +-----------------------------------+
==> 1 row
==> 4 ms
```

Conclusion

Nous vous avons brièvement présenté Neo4j, et seulement du point de vue de quelques requêtes Cypher simples. Sachez que Neo4j est très rapide, même avec des millions de nœuds et de relations, et que son langage est très complet. Vous disposez également de bibliothèques clientes pour différents langages. Le client pour Python s'appelle py2neo.

Partie III

Mettre en œuvre une base NoSQL

Cette partie est consacrée à la mise en œuvre des bases de données NoSQL. Dans un premier temps, nous verrons si le passage à une base NoSQL se justifie, et si c'est le cas, vers quel type de base de données se tourner. Nous aborderons ensuite la modélisation des données et déterminerons si elle s'avère nécessaire en NoSQL. L''installation et le déploiement des moteurs NoSQL seront également étudiés, ainsi que leur supervision. Pour finir, des exemples concrets de mise en œuvre de bases NoSQL seront proposés à travers une étude de cas.

13

Quand aller vers le NoSQL et quelle base choisir ?

L'adoption d'un moteur NoSQL ne se fait pas sur un coup de tête, uniquement parce que la technologie a l'air intéressante. Les données sont dans la majorité des cas le centre d'un système informatique, que l'on nomme justement un système d'information. Le choix d'un système de gestion de données ne doit pas se faire selon les tendances du moment, mais par rapport aux besoins et aux contraintes du métier. Il est important d'effectuer le bon choix et pour les bonnes raisons.

Aller ou non vers le NoSQL

Comment les données vont-elles être utilisées ?

Les moteurs NoSQL présentent certains avantages : souplesse du schéma, typage dynamique, proximité avec les langages clients, montée en charge facilitée par une approche horizontale avec sharding automatique, etc., ce qui permet une gestion plus aisée du système d'information. Mais cela impose aussi des contraintes. En effet, la relative souplesse du schéma et la proximité avec le langage client orientent les moteurs NoSQL vers une organisation des données plutôt par application que vers une centralisation et une utilisation partagée.

Le modèle des bases de données relationnelles sépare la conception logique et les structures physiques. Les données atomiques et fortement typées sont très structurées. Des contraintes et des règles métier au niveau du serveur (par les vues, les déclencheurs et les procédures stockées) sont mises en place. Ceci permet de gérer les données en amont, presque indépendamment des applications clientes, et donc de partager et réutiliser ces données pour toute l'entreprise.

C'est une force incontestable. Mais qui va de pair avec quelques contraintes : un peu moins de souplesse au niveau de l'évolution du schéma et l'obligation d'une phase d'analyse plus poussée au cours de laquelle les données sont prédéfinies avec soin.

Cela dit, un moteur orienté documents comme MongoDB, ou CouchDB, regroupe ses données dans une unité (le document JSON) qui correspond aux besoins d'une application (toutes les données nécessaires à une page web, à un formulaire, à une commande, à une action). L'unité de travail n'est plus une entité découpée selon la logique de la donnée, comme c'est le cas pour les relations dans le modèle relationnel, mais selon la logique de l'application, du besoin du client. C'est donc une approche plus orientée vers le développeur, et moins vers une donnée considérée comme centrale, qui fait la loi. L'approche tranche avec les habitudes des DBA, mais pourquoi pas. Toutefois, cela implique d'avoir bien saisi cette nuance et d'en tirer toutes les conclusions, notamment en ce qui concerne la réutilisation des données dans les différentes étapes du traitement. Par exemple, une application de gestion qui manipule des produits, des commandes, puis un suivi de commandes, et réutilise des données analytiques pour des besoins décisionnels et des tableaux de bord, se doit d'avoir des données accessibles selon plusieurs points d'entrée, notamment pour des calculs statistiques après coup.

Comme nous l'avons vu, les moteurs NoSQL organisent leurs données selon un identifiant de ligne, une clé, appelée « primaire » dans le modèle relationnel. Afin de pouvoir faciliter le partitionnement des données, il vaut mieux que ces dernières soient accédées exclusivement par la clé, au moins dans la plupart des cas. Le besoin de recherche à l'intérieur de la valeur qui est stockée avec cette clé dépend du type de moteur NoSQL. Les moteurs orientés colonnes et orientés documents vous permettent de créer des index secondaires, mais cette approche ne présente pas la souplesse et la solidité des moteurs relationnels. On reste dans une vision organisée principalement selon la clé. En ce qui concerne les moteurs de paires clé-valeur, comme Redis ou Riak, la partie valeur est théoriquement opaque. Le moteur n'a pas de vision de cette partie, qui peut être un objet binaire, une classe sérialisée, des structures diverses. Le moteur ne possède pas les outils permettant d'interpréter et de donner sens à cette donnée. En théorie, il est donc impossible d'indexer ou d'effectuer une recherche dans la partie valeur. C'est bien entendu à moitié vrai. En effet, Redis permet différents types de données pour la partie valeur et optimise l'accès, par exemple, à un ensemble ou un ensemble ordonné. Riak, quant à lui, implémente depuis peu des index secondaires.

Il existe deux approches traditionnelles pour effectuer des recherches dans ces moteurs :

- dupliquer la donnée à rechercher dans une autre structure où elle devient la clé, et la valeur est la référence de la clé de la collection d'origine ;
- lorsqu'on manipule du texte, indexer la partie valeur avec un moteur de recherche en texte intégral, comme Lucene ou Solr, (voir chapitre 12, section « ElasticSearch »).

Intérêt d'ElasticSearch

Pour les schémas simples et des besoins de grande rapidité, il est intéressant d'opter pour une solution Redis ou Riak + ElasticSearch.

Les avantages des moteurs relationnels

Dès que les données sont fortement structurées et que les besoins de recherche sont divers sur plusieurs attributs, avec la nécessité de s'adapter à tous types de requêtes provenant des utilisateurs, les capacités déclaratives et ensemblistes du langage SQL alliées aux techniques d'indexation des moteurs SQL offrent d'excellentes performances, surtout par leur capacité à utiliser un moteur d'optimisation pour établir une stratégie procédurale au niveau du serveur. Même si on les considère du point de vue du développement rapide, les moteurs relationnels sont très intéressants, car l'optimisation peut être effectuée en créant des index ou des vues matérialisées sans avoir à changer le code client.

On pourrait résumer les choses ainsi : si vous avez besoin d'une souplesse au niveau du schéma, +1 pour le NoSQL ; si vous avez besoin d'une souplesse au niveau de la richesse des requêtes, +1 pour les moteurs SQL.

Une autre conséquence de la diminution de la modularité des données imposées par l'organisation clé-valeur est la réutilisation des structures, comme les documents, organisées par d'autres clés. Par exemple, si vous stockez des commandes de produits dans des documents MongoDB dont la clé est un numéro de commande, vos choix seront ensuite limités si vous souhaitez organiser vos données selon le client (pour suivre de son compte) ou selon le produit (pour gérer les stocks). Vous serez sans doute obligé de dupliquer la donnée dans d'autres structures. Pour cette raison, les bases NoSQL ne sont pas des moteurs intéressants pour des applications de gestion classiques, pour lesquelles le modèle relationnel est plus intéressant.

En résumé, si vous avez besoin d'un système d'information dans lequel les données doivent être organisées d'une façon centrale, partagées entre différentes applications, recherchées selon de multiples critères, il vaut mieux vous orienter vers un moteur relationnel. En revanche, si vos cas d'utilisation sont dédiés à peu d'applications bien ciblées, et que vous avez un modèle où la plupart des accès seront faits selon une clé bien définie, le paradigme NoSQL est intéressant.

Que doit-on stocker ?

On peut se poser la question autrement : de quelle nature sont mes données ? S'agit-il de données fortement structurées et typées avec une certaine stabilité ou plutôt des données déstructurées ou semi-structurées, dont les attributs varient fortement selon les cas ? Un produit de même type ou une facture présente des attributs assez stables. En revanche, un article de blog n'a pas beaucoup d'attributs, ceux-ci pouvant être assez variables : un texte plus ou moins long, des images, de la vidéo, du son…, éléments difficiles à gérer dans un moteur relationnel. Cette structure est-elle stable ou est-il prévu qu'elle soit fortement modifiée ? Un moteur orienté documents peut être une bonne solution : en versionnant dès le départ le document stocké, le modèle peut évoluer sans changement de structure dans le moteur, et avec une gestion conditionnelle dans le programme client.

On peut aussi réfléchir selon le type des données. Des cas d'utilisation comme les blogs ou les réseaux sociaux représentent des structures de données plutôt orientées texte. Les applications de gestion, qui sont des cas d'utilisation typiques des moteurs relationnels, sont plus orientées autour des données chiffrées, les parties texte étant limitées aux libellés, références, parfois aux descriptions. Dans une application de gestion, les données chiffrées doivent être retrouvées selon plusieurs critères et des calculs doivent être effectués selon différents niveaux de regroupement.

Une base de données relationnelle est vraiment plus adaptée dans ce cas. Ce sont finalement des choses que l'on représente déjà naturellement de manière tabulaire dans un tableur, en créant des références entre cellules. La base de données relationnelle en est l'extension beaucoup plus puissante, mais qui reste un peu dans le même concept. Une base de paires clé-valeur est l'extension des listes de données, et un moteur orienté documents est l'extension d'une structure hiérarchique de données orientées texte, qu'on peut aussi représenter plus facilement dans du XML que dans une structure tabulaire.

La différence d'approche avec le relationnel

Le NoSQL implique également une différence au niveau de la phase de conception. Un projet informatique traditionnel incluant la gestion de données relationnelles comporte une phase d'architecture et d'analyse fonctionnelle spécifique. La phase de modélisation est très importante dans un tel projet, même s'il est défini comme agile. Il est vital de comprendre le modèle de données à l'avance, et il s'agit souvent d'un travail effectué par un spécialiste. L'approche d'un moteur NoSQL est plus informelle, elle se plie beaucoup plus aux besoins de l'application cliente elle-même. Le choix d'un moteur NoSQL est souvent motivé par des raisons non fonctionnelles, mais sur des bases techniques : volumétrie, performances, distribution. Mais comme nous l'avons vu, cela dépend du type de moteur NoSQL. Une simple base clé-valeur, voire une base orientée documents, entre dans cette catégorie « informelle ». Un moteur orienté colonnes comme Cassandra, en raison de la centralité plus forte de ses données et d'une plus grande formalisation des structures, se rapproche plus d'un moteur relationnel du point de vue de la phase de conception.

Une autre question est celle du niveau de compétence des équipes. Une base NoSQL est plus accessible aux équipes de développeurs habitués à des langages objet ou impératifs. Il vaut mieux finalement faire du NoSQL que du SGBDR mal conçu en utilisant un ORM *(Object-Relational Mapping)* comme Hibernate ou Entity Framework.

Le problème des compétences

Les compétences sont un élément important de la prévision de montée en charge. Monter en charge avec un SGBDR nécessite des compétences dans ce domaine : tuning, optimisation de la structure, des index et du code SQL, mise en œuvre de mécanismes comme le partitionnement de table ou les vues matérialisées. Obtenir de bonnes performances demande une attention constante à la base de données, depuis les premiers moments de la phase de conception et tout le long de la vie de l'application. Chaque étape de la mise en œuvre d'une base de données relationnelle est importante pour les performances (le modèle de données, le code SQL, l'indexation et l'implémentation physique). Les erreurs sont difficiles à corriger après coup.

De plus, un des principaux critères de qualité d'un moteur relationnel est son moteur d'optimisation, c'est-à-dire le sous-système du moteur relationnel chargé de transformer la requête déclarative en un plan d'exécution procédural. Beaucoup de personnes qui sont passées du relationnel au NoSQL, déçues par les performances du premier, citent une utilisation de MySQL. Or, MySQL est sans doute un excellent moteur, mais il n'atteint pas la qualité et les performances des SGBDR commerciaux, ou même de PostgreSQL si on reste dans le monde du libre. Le problème peut alors devenir un problème de licence : on se retrouve à devoir payer très cher un moteur qui peut offrir

d'excellentes performances à la condition que les équipes qui le mettent en œuvre en comprennent les subtilités, ce qui a également un certain coût en termes de temps et de formation. Dans ce cas, même si dans l'absolu l'utilisation du NoSQL ne se justifie par réellement, il peut constituer un choix plus simple et moins coûteux.

Finalement, monter en charge avec un moteur NoSQL nécessite peu de compétences logicielles. Comme il s'agit principalement d'une distribution de traitement, il « suffit » d'ajouter de nouvelles machines et de les configurer à l'identique des machines déjà installées pour qu'elles soient prises en compte par des mécanismes comme le sharding ou l'algorithme MapReduce.

Intégré dans la plupart des moteurs NoSQL, MapReduce représente l'algorithme de choix pour effectuer après coup des traitements sur les données avec des performances raisonnables, l'équivalent des calculs qu'on peut réaliser avec le langage SQL, comme les agrégats. Mais, sur des tailles de données raisonnables, disons par exemple jusqu'à quelques téraoctets, MapReduce ne peut rivaliser en performance pour les calculs avec un moteur relationnel ou un système décisionnel. MapReduce offre, nous l'avons dit, des temps de réponse raisonnables, mais il n'est pas conçu pour retourner des réponses dans des conditions de temps réel.

Quels sont les besoins transactionnels ?

Sauf rares exceptions, les moteurs NoSQL n'ont pas de notion de transaction qui permettrait de rendre atomique des modifications opérées sur plusieurs lignes ou plusieurs documents. Des moteurs comme MongoDB permettent de simuler des relations entre documents en stockant la clé d'une autre collection (un type `DBRef` dans MongoDB), sinon le traitement est opaque pour le moteur et géré uniquement par la logique du code client, sans aucun moyen de rendre ces opérations sur plusieurs données atomiques. Les besoins transactionnels complexes sont finalement rares dans beaucoup d'applications, et dans les cas où ce besoin se fait sentir (application financière, dernier stade du panier d'achat, etc.), des moteurs relationnels ou qui supportent pleinement l'acidité de la transaction (tels que Neo4j dans le monde NoSQL) devront être choisis. Les moteurs orientés colonnes permettent à la lecture et à l'écriture de déterminer un niveau de cohérence, ils constituent donc un choix souple. Cependant, ils sont à privilégier pour des besoins de volumes importants, car ils sont conçus pour être distribués. Installer HBase ou Cassandra sur une seule machine, par exemple, n'a pas vraiment de sens.

Résumé des cas d'utilisation

En résumé, quand est-il une bonne idée d'utiliser une base NoSQL ? Prenons d'abord la question du point de vue des atouts du non relationnel :

- les données que vous devez gérer sont peu structurées, et les structures sont plutôt changeantes. Il s'agit par exemple de texte plus ou moins long, de données saisies par les utilisateurs sans canevas trop contraignant. Vous allez naturellement vous tourner vers une base orientée documents ;
- vous devez privilégier les performances d'écriture et de restitution, et vous assurer que l'accès aux données se fasse toujours en dessous de quelques millisecondes. Une base en mémoire comme Redis sera alors un bon choix ;

- vous devez stocker et manipuler de grandes quantités de données, par exemple plusieurs téraoctets. Un moteur distribué sera alors un choix naturel ;
- vous accéderez à vos données principalement par un seul point de recherche, qui constituera la clé de vos documents ou de vos paires clé-valeur. Par exemple, vous voulez afficher sur une page toutes les données d'un utilisateur. Un document JSON qui stocke ces informations et que vous requêterez par l'identifiant de l'utilisateur (la clé) sera idéal.

En revanche, le NoSQL présente quelques désavantages qui feront que vous privilégierez un moteur relationnel dans certains cas :

- vous stockez et manipulez des données très structurées, sur lesquelles vous voulez appliquer différents calculs, du reporting selon plusieurs axes, et que vous voulez pouvoir chercher selon différents critères ;
- vos données sont manipulées depuis plusieurs applications clientes, et vous voulez pouvoir assurer la qualité de ces données en appliquant des contraintes au niveau du serveur ;
- vous effectuez des recherches ou des traitements complexes, en masse, sur des ensembles de données. Dans ce cas, les moteurs relationnels vous permettent de manipuler aisément, de façon ensembliste, vos données avec du code qui s'exécute sur le serveur.

Une des grandes forces des moteurs relationnels reste la possibilité de traitement du côté du serveur amenée par le langage SQL.

Choix par rapport au type d'application

On peut aussi considérer le choix d'un moteur relationnel ou NoSQL globablement par le type d'application de données développée. Voici quelques exemples d'applications et du raisonnement concernant le choix d'un moteur.

- **Blog** – un blog est principalement constitué de texte (articles, commentaires) et de listes (mots-clés, catégories). Les articles de blog sont principalement appelés par un critère, la date de l'article, auquel s'ajoute éventuellement l'identifiant de l'auteur et la catégorie. Un moteur NoSQL orienté documents, comme MongoDB ou CouchDB, est particulièrement indiqué pour ce genre d'application, auquel on peut ajouter les applications de messagerie, les bases de connaissance, les magazines. Il n'y a aucun avantage à utiliser un moteur relationnel dans ce type d'application. Pour les fonctionnalités de recherche, les index des moteurs relationnels ne seraient d'aucune utilité sur du texte long. L'indexation en plein texte des documents JSON avec un outil comme ElasticSearch est donc la meilleure solution.
- **Données de gestion, ERP** – les données de gestion sont souvent très structurées, avec des données atomiques bien définies, des besoins de manipulation à partir de plusieurs briques logicielles clientes et des besoins de recherches multicritères et de manipulations complexes. De plus, les volumes de données restent raisonnables et nécessitent rarement de distribuer le traitement. Toutes ces raisons font qu'un SGBDR reste le meilleur choix dans ce genre de cas.
- **Reporting et analyse OLAP** – les besoins de reporting impliquent de faire des recherches à travers de grandes quantités de données bien structurées, afin d'effectuer des calculs d'agrégation sur plusieurs axes d'analyse. Un moteur relationnel, ou mieux, un moteur OLAP ou une solution de reporting qui manipule directement des structures multidimensionnelles est

à privilégier au NoSQL dans ce cas, sauf si le volume des données devient très important, auquel cas vous aurez besoin de distribuer le traitement. Un moteur orienté colonnes, comme HBase ou Cassandra est alors une bonne solution, quitte à stocker des données précalculées et dupliquées selon plusieurs clés correspondant aux axes d'analyse, et à profiter de traitements en MapReduce.

- **GED, Gestion électronique de documents** – la gestion électronique de documents consiste à dématérialiser les documents et à les classer dans le système informatique. Il n'y a pas forcément de gain apporté par les SGBDR dans ce modèle, vous pouvez utiliser un moteur NoSQL. Les bases de données en paires clé-valeur peuvent contenir des documents binaires dans la partie valeur, et un moteur comme MongoDB comporte une fonctionnalité de stockage de fichiers nommée GridFS. Ceci dit, comme en ce qui concerne les SGBDR, il est souvent plus efficace d'organiser les fichiers sur un système de fichiers partagé et de référencer seulement les liens dans la base de données. Les moteurs relationnels comportent souvent un type de données nommé DATALINK ou FILESTREAM qui permet de stocker les binaires sur un emplacement disque tout en conservant une cohérence transactionnelle. C'est une fonctionnalité utile si vous optez pour un moteur relationnel.
- **E-commerce** – l'e-commerce est typiquement un domaine où les moteurs NoSQL peuvent être utilisés en parallèle avec les SGBDR. La partie catalogue, qui comporte des données semi-structurées en lecture seule, peut bénéficier du stockage en documents JSON ou des performances apportées par des moteurs en mémoire comme Redis. Une indexation avec ElasticSearch ajoute les fonctionnalités de recherche dans le catalogue. Le panier d'achat, s'il doit être distribué pour pouvoir monter en charge facilement, peut être conservé dans un moteur distribué comme Riak, Cassandra ou MongoDB. Quant à la partie validation des achats, la nécessité de la générer transactionnellement vous poussera à privilégier un moteur relationnel, quitte à partitionner les données sur plusieurs machines manuellement en faisant un choix à partir du client, par exemple avec un modulo du hashcode de votre identifiant client calculé dans votre application cliente.
- **Logistique** – les applications logistiques peuvent bénéficier du NoSQL : suivi d'un colis ou d'un container via un document JSON dans lequel sont listées toutes les étapes du transport, recherche d'un chemin optimal en utilisant une base de données orientée graphes comme Neo4J, stockage des manifestes et listes de colisage dans des documents JSON… La logistique est un domaine d'application où il est intéressant de mettre en place plusieurs solutions de gestion de données selon les besoins. Une base de données relationnelle peut rester la colonne vertébrale des données par sa gestion centralisée de ces dernières bien structurées, et des moteurs NoSQL peuvent être utilisés pour offrir des fonctionnalités spécifiques.
- **Données spatiales** – plusieurs moteurs NoSQL commencent à implémenter des fonctionnalités de calcul spatial, comme les calculs de distance entre les points de MongoDB, ou le support naissant de la spécification GeoJSON (http://www.geojson.org/, une définition de document JSON comportant des objets spatiaux comme des points ou des polygones) dans CouchBase Server.
- Il n'y a rien toutefois qui soit suffisamment avancé dans le monde NoSQL à l'heure actuelle pour développer de réelles applications spatiales. Il faut donc rester pour l'instant avec des outils qui s'appuient sur les SGBDR, comme l'extension PostGIS de PostgreSQL, qui supporte GeoJSON.

Quelle base choisir ?

Comparaison des principales bases NoSQL

Le monde NoSQL est une nébuleuse de moteurs relativement différents, qu'il n'est pas toujours aisé de rassembler sous la même bannière. Si vous décidez d'opter pour un moteur NoSQL, il convient de choisir celui qui sera le plus adapté à vos besoins, chacun des moteurs répondant à des cas d'utilisation très différents.

Il nous a paru utile de récapituler sous une forme synoptique les principales caractéristiques des moteurs NoSQL dont nous avons parlé jusqu'à présent. Nous avons essayé de résumer les critères de choix de chaque moteur, la structuration des données qu'il permet, sa façon de gérer la problématique de la cohérence, s'il permet ou non la distribution des données sur plusieurs nœuds, en quel langage il est écrit, son mode de licence, le protocole de communication utilisé et les points forts du moteur. Cela vous permettra de disposer d'un tableau de référence pour une première analyse selon vos besoins. Voici donc la liste résumée des moteurs.

HBase

À choisir pour : HBase est un choix intéressant uniquement si vous prévoyez de gérer un volume de données très important. Basé sur Hadoop, il permet de distribuer les données en utilisant le système de fichiers distribué HDFS *(Hadoop Distributed File System)* d'Hadoop. Cela n'a donc pas de sens de vouloir utiliser HBase sur un système non distribué. De plus, sa mise en œuvre est relativement complexe.

Types de données manipulées : base de données orientée colonnes. Les données sont organisées selon des clés de ligne *(row key)*, puis en familles de colonnes, puis en colonnes. La cellule est composée d'un nom de colonne, de la valeur et d'un timestamp. La valeur est simplement stockée sous forme d'octets, il est inutile de définir des types de données ou de prédéfinir les colonnes, seules les familles de colonnes doivent être prédéfinies.

Maintien de la cohérence : garantie ACID sur une ligne (c'est-à-dire plusieurs familles de colonnes) depuis HBase 0.92. Cela veut dire que des commandes qui appliquent des modifications sur plusieurs lignes ne constituent pas une opération atomique, mais une suite de modifications de lignes individuellement atomiques, qui vont retourner un état de succès ou d'erreur pour chaque ligne.

Mode de distribution : basé sur Hadoop et HDFS, avec maître centralisé. Un cluster HBase est constitué d'un maître (`HMaster`) qui maintient les métadonnées du cluster et gère des serveurs de région (`RegionServer`). Chaque région contient une partie des données.

Développé en : Java. Toute la pile Hadoop, HDFS et HBase est en Java.

Licence et support : Apache 2.0. HBase est un projet de la fondation Apache (http://hbase.apache.org/license.html). La société américaine Cloudera (http://www.cloudera.com/) distribue une édition de Hadoop et HBase avec support nommée Cloudera Enterprise.

Protocole : REST et Thrift. Il existe également une API Java native et une interface Avro.

Points forts : à utiliser pour le Big Data, très bonne montée en charge horizontale, solidité de la conception et excellente tolérance au partitionnement. HBase est utilisé par des acteurs comme

Facebook pour stocker tous les messages de ce réseau social, ce qui représentait en 2009 plus de 150 téraoctets de nouvelles données par mois (https://www.facebook.com/UsingHbase). Sa popularité fait qu'il est en constante amélioration de par le développement de Cloudera et les contributions de la communauté. Par exemple, Facebook a apporté de nombreuses améliorations au moteur.

Cassandra

À choisir pour : choix intéressant pour de grands volumes de données et les besoins de bases distribuées, hautement disponibles et décentralisées, sur de multiples data centers. Comme pour HBase, Cassandra n'est pas un choix intéressant pour des volumes de données moyens ou des systèmes non distribués.

Types de données manipulées : base de données orientée colonnes. Données organisées dans un keystore (l'équivalent d'une base de données) par famille de colonnes (l'équivalent d'une table) selon une clé et des colonnes. La cellule est composée d'un nom de colonne, de la valeur et d'un timestamp. Les familles de colonnes doivent être prédéfinies, mais les colonnes dans une famille ne font pas l'objet d'un schéma défini. Contrairement à HBase, Cassandra a une clé par famille de colonnes, ce qui le rapproche plus d'un modèle de base de données relationnelle. Même si Cassandra stocke ses données sous forme de tableaux d'octets, l'abstraction amenée par CQL 3 le rend très proche au ressenti d'un moteur relationnel.

Maintien de la cohérence : la cohérence en écriture et en lecture sont paramétrables. Par exemple, on peut forcer une écriture, ou une lecture, à être validée sur plusieurs réplicas avant d'être considérée comme effectuée. À partir de Cassandra 1.1, l'ACIDité est garantie par ligne.

Mode de distribution : décentralisé. Chaque nœud est indépendant et il n'y a pas besoin de serveur maître. Les connexions utilisateurs peuvent se faire sur n'importe quel nœud, qui se chargera de rediriger le client vers le nœud qui contient les données souhaitées. La répartition est réalisée par vnodes ou, historiquement, par hachage consistant.

Développé en : Java.

Licence et support : Apache 2.0. Cassandra est un projet de la fondation Apache. La société américaine DataStax (http://www.datastax.com/) maintient une édition de Cassandra, nommée DataStax Enterprise, gratuite au téléchargement avec une option d'abonnement avec support.

Protocole : l'API Cassandra est de type natif pour CQL. Une interface Thrift existe mais elle est dépréciée.

Points forts : implémentation solide, système décentralisé, souplesse de configuration, cohérence paramétrable. Ne nécessite pas un système de fichiers distribué comme HBase. Cassandra est un produit mature, largement utilisé et très populaire, c'est une excellente solution pour bâtir un système de gestion de données volumineux et décentralisé.

CouchDB

À choisir pour : CouchDB est la base de données du Web. Son orientation documents et son interface REST excellent pour construire des projets web ou à destination des terminaux mobiles. CouchDB est à préférer pour les besoins où la richesse fonctionnelle l'emporte sur les grands volumes et la disponibilité. Pour les besoins de grandes performances, voir du côté de Couchbase Server, le successeur de CouchDB.

Types de données manipulées : document JSON.

Maintien de la cohérence : cohérence finale *(eventual consistency)*. À la base, CouchDB n'étant pas distribué, la cohérence locale est gérée par un verrouillage optimiste : CouchDB maintient automatiquement un numéro de version sur chaque document. Pour modifier un document existant, il faut indiquer son dernier numéro de révision, sinon la mise à jour est refusée.

Mode de distribution : pas de distribution native. Un système de réplication des données permet d'échanger les modifications d'une base avec un autre serveur, mais cela doit être géré manuellement. Pour une version nativement distribuée, voir Couchbase Server.

Développé en : Erlang, un langage développé à l'origine par la société Ericksson et qui est conçu spécialement pour le traitement parallèle et l'informatique distribuée.

Licence et support : Apache 2.0. CouchDB est un projet de la fondation Apache.

Protocole : REST exclusivement. Il n'y a pas d'interface rapide et plus bas niveau.

Points forts : grande richesse fonctionnelle au niveau du serveur, vues, filtres dans des documents de design et incorporation possible de traitements complexes sur le serveur, ce qui peut l'amener à devenir également un vrai serveur applicatif. La simplicité de mise en œuvre est aussi un des points forts de CouchDB, qu'on a étiqueté de « base de données du Web ». Fonctionnalités de réplication et de surveillance des changements simples mais solides et bien faites.

Couchbase Server

À choisir pour : Couchbase est un moteur décentralisé paires clé-valeur et document JSON. C'est un excellent moteur pour des besoins de stockage simple ou de type document avec un besoin d'élasticité dans la montée en charge.

Types de données manipulées : paire clé-valeur et document JSON.

Maintien de la cohérence : cohérence forte, simplement parce que l'écriture et la lecture sont réalisées sur le même nœud, qui est maître des données. Les réplicas existent pour la redondance uniquement.

Mode de distribution : décentralisé. Chaque nœud est indépendant et il n'y a pas besoin de serveur maître. Les bibliothèques clientes qui utilisent le concept de smart client récupèrent en cache local la topologie du réseau et adressent directement le bon nœud lorsque ces clients font un get d'une clé. La répartition est faite par vbucket, un type de vnode.

Développé en : C++ et Erlang.

Licence et support : Apache pour l'édition communautaire, et commerciale.

Protocole : memcached pour l'accès aux clés, REST pour l'accès aux vues.

Points forts : excellentes performances dans l'accès aux clés, richesse fonctionnelle au niveau du serveur à travers les vues, élasticité de la montée en charge.

MongoDB

À choisir pour : tout type de besoin de stockage de documents, c'est-à-dire de données structurées : sérialisation d'objets, pages web, formulaires de saisie, informations d'applications. La représentation en documents permet d'utiliser MongoDB pour des structures proches de ce qui

est stocké dans une base relationnelle, ou des structures hiérarchiques, avec une plus grande souplesse dans le schéma de données. Un système d'indexation secondaire permet d'effectuer des recherches dans les données et de récupérer des documents par d'autres points d'entrée que la clé.

Types de données manipulées : BSON, un format JSON «binarisé» pour être plus compact.

Maintien de la cohérence : cohérence finale à travers les réplicas. Les écritures ont une option *(write concern)* pour indiquer qu'elles sont validées si elles ont écrit sur un certain nombre de réplicas ou sur une majorité (N/2+1). La durabilité est optionnelle à travers l'activation d'un journal *(Write-ahead log)*. Une mise à jour est atomique par document, il n'y a pas de transaction multidocument possible.

Mode de distribution : centralisé. Autosharding et réplication automatique. Les métadonnées des shards sont centralisées sur un serveur de configuration, qui peut lui-même être répliqué pour fournir de la redondance. La connexion client se fait aussi sur un serveur maître, qui redirige la requête vers le bon serveur de shard. Ce serveur maître peut lui aussi être redondant.

Développé en : C++.

Licence et support : GNU AGPL v 3.0, licence Apache pour les pilotes. La licence AGPL permet de s'assurer que les contributions retournent à la communauté. Des licences commerciales peuvent être acquises auprès de MongoDB Inc. (http://www.mongodb.org/display/DOCS/Licensing).

Protocole : natif, question-réponse sur un socket.

Points forts : moteur solide, bonnes performances. Simplicité d'utilisation du point de vue du développement client et bonne intégration des pilotes dans les langages. Support commercial par la société qui développe le produit, MongoDB Inc. Ses capacités de sharding et de réplication automatiques permettent de monter en charge horizontalement au fur et à mesure de l'augmentation des besoins.

Riak

À choisir pour : Riak est un entrepôt de paires clé-valeur bien conçu, solide, performant et qui monte très bien en charge. C'est un excellent choix pour un moteur qui doit à la fois être distribué et offrir une latence faible, sur des systèmes qui doivent être prêts à monter en charge très rapidement et de façon automatique.

Types de données manipulées : paires clé-valeur. La valeur est normalement opaque, mais elle peut être indexée si c'est du JSON.

Mode de distribution : décentralisé par hachage consistant, à la manière de Dynamo d'Amazon. Pas de serveur maître et chaque nœud est indépendant. Les connexions utilisateurs peuvent s'effectuer sur n'importe quel nœud, qui se chargera de rediriger le client vers le nœud contenant les données souhaitées.

Développé en : Erlang, un langage développé initialement par la société Ericksson et qui est spécialement adapté au traitement parallèle et à l'informatique distribuée.

Licence et support : Apache 2.0. Riak est développé par la société Basho (http://basho.com/), qui en distribue deux versions commerciales : Riak Enterprise, qui étend les fonctionnalités de Riak en ajoutant une réplication sans maître et la supervision en SNMP, et Riak Cloud Storage pour l'utilisation de Riak dans le cloud.

Protocole : Protobuf et REST. Protobuf est bien sûr à privilégier pour de meilleures performances. L'interface REST est intéressante pour les opérations d'administration ou un accès aux données par des applications mobiles.

Points forts : une des implémentations qui se rapprochent le plus de Dynamo d'Amazon et qui reprend ses solutions techniques : hachage consistant, *read repair*, *hinted handoffs*, etc. Support commercial par la société Basho qui développe le produit. Offre de bonnes performances et de bonnes capacités de montée en charge.

Redis

À choisir pour : Redis est un excellent choix pour maintenir des données en mémoire pour un accès en temps réel très rapide. C'est une forme de remplacement d'un cache tel que memcached pour offrir une plus grande richesse fonctionnelle et une manipulation de structures de données plus riches.

Types de données manipulées : à la base, une chaîne binary-safe dans laquelle on peut donc stocker et manipuler des valeurs numériques et binaires. À partir de ce type de donnée de base, Redis offre des listes, des ensembles, des ensembles triés et des tables de hachage. Ces structures permettent de gérer beaucoup de besoins de manipulations de données, et ces structures sont optimisées en mémoire.

Maintien de la cohérence : pas de notion particulière de cohérence. Une seule écriture est bien sûr atomique. Une pseudo transaction peut être définie pour exécuter plusieurs instructions en une seule fois sur un mode de mise en attente et d'empilement des commandes, et d'exécution groupée à la fin.

Développé en : ANSI C.

Licence et support : BSD *(Berkeley Software Distribution Licence)*, une licence libre peu contraignante qui permet d'utiliser, de modifier ou de distribuer Redis sans restriction, que ce soit pour des besoins libres, propriétaires, gratuits ou commerciaux. Il n'y a pas de support officiel. Redis est principalement développé par Salvatore Sanfilippo et Pieter Noordhuis, engagés par VMWare pour développer Redis à plein temps en licence libre.

Protocole : natif, mode question-réponse à travers un socket.

Points forts : extrême rapidité, solidité et intelligence de la construction, richesse fonctionnelle du langage pour la manipulation des données, opérations sur les ensembles.

En résumé

Terminons ce chapitre par un tableau comparatif synthétique qui, pour chaque moteur traité dans ce livre, offre quelques pistes d'utilisation.

Tableau 13-1. **Comparatif des bases NoSQL**

Moteur NoSQL	Type de données	Intérêt	Utilisation
HBase	Colonnes	Big Data	Excellentes performances sur de très grands volumes.
Cassandra	Colonnes	Big Data	Montée en charge linéaire sur des grands volumes de données.
CouchDB	Documents JSON	Fonctionnalités	Développements web sans besoin de très bonnes performances.
Couchbase	Documents JSON	Performances	Montée en charge linéaire sur des grands volumes de données et richesse fonctionnelle.
MongoDB	Documents JSON	Sharding et réplication	Montée en charge facilitée sur des données semi-structurées.
Riak	Clé-valeur	Décentralisé	Excellentes performances et montée en charge très aisée en ajoutant des nœuds.
Redis	Clé-valeur	Performances	Entre le système de cache et la base de données. Assure d'excellentes performances sur des traitements de données simples.

Conclusion

Le choix d'un moteur de base de données se fait à partir des besoins, et certainement pas à partir d'un a priori technologique. Autant que possible, ce choix doit aussi se baser sur des tests réalisés en interne et prenant en compte les contraintes du business. C'est à la fois un choix décisif, car il est difficile de revenir en arrière lorsque toutes vos données sont gérées dans un système, mais c'est aussi un choix qui n'est pas forcément fatal, car il est possible d'adapter son traitement, jusqu'à un certain point, au moteur choisi. Les axes de choix comportent les fonctionnalités, les performances et la solidité du produit. Il est par exemple possible de faire le choix, pour un système de production, d'un moteur NoSQL encore jeune, mais il faut alors pouvoir contourner les problèmes rencontrés lors de l'utilisation. Nous pensons notamment à des outils, comme Couchbase Server, dont nous n'avons pas parlé dans ce livre, parce qu'ils sont trop jeunes, mais qui sont parfois mis en production dans des entreprises qui essuient les plâtres. Donc, votre choix devra se faire selon les axes que nous avons indiqués dans ce chapitre, mais aussi après des tests aussi sérieux que possible pour vous assurer que le moteur réponde à vos besoins, et qu'il est capable de tenir la charge solidement, comme vous le souhaitez. Souvenez-vous en plus d'une chose : dans le monde NoSQL, un moteur correspond souvent à une application cliente, vous pouvez donc tout à fait mettre en place plusieurs solutions de gestion de données selon vos besoins, et établir des fertilisations croisées entre les SGBDR, les moteurs NoSQL et des outils comme ElasticSearch pour ajouter les fonctionnalités voulues.

14

Mettre en place une solution NoSQL

Même si les moteurs NoSQL sont plutôt organisés selon une optique de développement rapide, et que l'établissement d'un schéma préliminaire n'est pas nécessaire, cela ne veut pas dire que leur mise en place doit être prise à la légère.

Dans ce chapitre, nous allons aborder quelques considérations d'architecture et de modélisation des données.

Architecture et modélisation

Comme évoqué dans le chapitre précédent, on peut se demander si la modélisation des données et l'architecture ont encore un sens dans le monde NoSQL. Certainement, et pour plusieurs raisons.

Tout d'abord, un moteur NoSQL est souvent utilisé en complément de données structurées dans un moteur relationnel. Pour Google, par exemple, MySQL est toujours un moyen important de stocker des données structurées, parallèlement à BigTable. Les questions d'architecture qui se posent ne concernent donc plus seulement la manière de modéliser les données, mais également l'endroit où les stocker. Traditionnellement, les bases relationnelles sont conçues en se posant des questions telles que :

- Doit-on séparer telles tables dans des schémas ou des bases de données différents ?
- Faut-il séparer des domaines (des colonnes) en colonnes plus atomiques ? Par exemple, faut-il dissocier le numéro, le type de voie et le nom de la voie dans une ligne d'adresse ?

- La structure des données est-elle susceptible d'évoluer, par exemple en ajoutant de nouveaux attributs ou en changeant des informations de référence ?

Ces questions traitent donc de la structure des données et de leur utilisation dans le cadre du projet ou de l'entreprise. Dans une conception incluant du NoSQL, les questions soulevées seront, par exemple, les suivantes :

- Devons-nous chercher les données et sur quels critères ?
- Quel est le rapport entre les lectures et les écritures ?
- Les données doivent-elles être opaques pour le moteur, ou faut-il qu'il puisse les traiter, par exemple en appliquant des opérations de MapReduce du côté du serveur ou des opérations de validation ?
- Faut-il effectuer des recherches sur les données, en d'autres termes, permettre des recherches sur des critères autres que la clé ?
- Les données doivent-elles être souvent réutilisées par d'autres applications ou d'autres services de l'entreprise ?
- Faut-il effectuer des calculs complexes ou lourds sur les données, tels que des agrégats ou des analyses temporelles ?
- Quelle est la latence acceptable d'accès aux données ? Faut-il privilégier les performances ou la capacité à supporter un fort volume ?

Ces questions tournent principalement autour de l'utilisation des données. La première question fondamentale pour décider si le choix doit se porter sur un moteur relationnel ou un moteur NoSQL est sans doute la question de la clé : se trouve-t-on devant un modèle où un accès via une clé, et seulement via cette clé, est possible, ou faut-il pouvoir interroger les données de façon complexe ? Plusieurs types de besoins se contentent parfaitement d'un accès par la clé : gestion de profils et de comptes utilisateurs, stockage de logs, informations de consommation ou d'utilisation de services, pour n'en citer que quelques-uns.

Les questions qui viennent après la détermination de la clé permettent de faire le choix d'un modèle de stockage, parmi les différentes formes proposées par les moteurs NoSQL. La structure relationnelle est une structure simple, en deux dimensions : lignes et colonnes, qui représentent bien des données fortement structurées. Le stockage dans un moteur NoSQL, tout en étant aussi très simple en apparence, permet de stocker des valeurs autrement plus complexes, ce qui procure une grande souplesse. Le modèle clé-valeur, notamment, allie simplicité et grande puissance car il répond à un grand nombre de cas d'utilisation, offre d'excellentes performances et l'avantage de la facilité de distribution. De plus, la valeur pouvant être n'importe quoi, il est facile d'y stocker tous types de données. Dès que la donnée doit cesser d'être opaque au moteur, il faut penser à la structurer en document, entendez du JSON.

Stocker du XML

Il est éventuellement possible de manipuler du XML, si c'est le format avec lequel vous travaillez nativement dans vos applications clientes, pour échanger des données par exemple. Le stockage se fera toujours en JSON, mais la conversion sera assurée par le code client, ou du code sur le

serveur, en utilisant la fonction `eval()` de MongoDB ou les vues dans CouchDB. Ainsi, dans les vues en CouchDB, vous pouvez manipuler du XML en utilisant l'interpréteur choisi, comme Python ou JavaScript.

Utiliser E4X pour générer du XML

En utilisant l'interpréteur JavaScript par défaut, vous pouvez aussi générer une sortie XML de vos documents JSON de façon très simple en utilisant E4X (ECMAScript for XML, la norme ECMA-357 : http://www.ecma-international.org/publications/standards/Ecma-357.htm). L'interpréteur JavaScript de MongoDB est le même que celui de CouchDB (SpiderMonkey), E4X est donc également supporté en MongoDB.

Vous pouvez trouver un exemple d'utilisation de vue générant du XML dans l'application Sofa développée pour les besoins du livre *CouchDB: The Definitive Guide*, disponible en ligne à cette adresse : http://guide.couchdb.org/. Il s'agit d'une application de blog totalement développée en CouchDB, dont le code source est maintenu sur GitHub à l'adresse suivante : https://github.com/jchris/sofa.

CouchApp

Sofa utilise CouchApp (http://github.com/couchapp/couchapp/), une suite d'outils destinés à faciliter l'intégration de documents de design et de pages de template, pour créer des applications web basées uniquement sur CouchDB. Un jeu d'outils plus moderne, nommé Erica, est maintenant disponible : https://github.com/benoitc/erica.

Sofa retourne des pages de blog formatées en HTML mais permet également de récupérer un flux XML en format de syndication Atom. Le code source de génération du flux est visible à l'adresse suivante : https://github.com/jchris/sofa/blob/master/lists/index.js et utilise la bibliothèque CouchApp (http://couchapp.org/). Nous reproduisons ci-après uniquement une fonction de ce fichier, qui suffira à illustrer l'utilisation très simple de E4X :

```
exports.header = function(data) {
    var f = <feed xmlns="http://www.w3.org/2005/Atom"/>;
    f.title = data.title;
    f.id = data.feed_id;
    f.link.@href = data.feed_link;
    f.link.@rel = "self";
    f.generator = "CouchApp on CouchDB";
    f.updated = rfc3339(data.updated);
    return f.toXMLString().replace(/\<\/feed\>/,'');
};
```

Cette fonction génère l'en-tête du flux Atom, en définissant une variable `f` de type XML (définie pour JavaScript dans la norme ECMA-357) et en retournant sa représentation en chaîne à la fin grâce à l'appel de sa méthode `toXMLString()`.

Conception pilotée par le domaine

La conception pilotée par le domaine (*Domain-Driven Design*, DDD) est une approche définie par Eric Evans dans son livre éponyme, *Domain-Driven Design*, paru chez Addison Wesley, dont nous vous recommandons la lecture. Il s'agit de principes de gestion de projets informatiques et de développement proches des méthodologies agiles qui se marient bien avec les projets NoSQL. Le livre original est plutôt épais, mais une version résumant la méthode est parue sous le titre *Domain Driven Design Quickly*, disponible librement au format PDF et que vous trouverez facilement en effectuant une recherche sur le Web.

Au niveau de la conception, Eric Evans insiste à juste titre sur la nécessité de bien comprendre le métier, le domaine pour lequel le développement est fait, et de modéliser ce métier en utilisant un vocabulaire commun entre le métier et la technique. Les phases d'établissement de ce modèle doivent impliquer également les développeurs qui vont travailler sur le code plus tard. Cela correspond bien avec l'utilisation des bases NoSQL, qui sont très orientées développement (parce que leur mise en œuvre est plus proche de la programmation pure et simple que de la modélisation et de l'administration). Les développeurs doivent parfaitement connaître le modèle et se sentir responsables de son intégrité. La structuration des données sera naturellement prise en charge par ces développeurs familiarisés avec le modèle et les besoins du métier.

Le modèle généré doit pouvoir être naturellement représenté en classes dans le langage de programmation. Ces classes représentent les objets du domaine. Certaines classes sont des entités. Elles ont une identité, c'est-à-dire un moyen d'identifier uniquement chaque membre, à l'aide d'un identifiant aussi simple que possible et immuable dans le temps, et non à l'aide de l'ensemble de ses attributs. Cela nous renvoie à notre clé dans un moteur NoSQL, une clé unique, préférablement technique (générée par le système, un UUID ou un entier, par exemple).

Le choix d'une clé technique

Que choisir comme clé dans une base NoSQL (ou dans une base relationnelle, la problématique étant exactement la même) ? Une clé naturelle, c'est-à-dire d'une valeur unique existant hors du système informatique (comme un numéro ISBN, un numéro de Sécurité sociale, un EAN, etc.) doit être évitée, car la clé doit rester stable, et les risques de changement des identifiants naturels sont grands. Par exemple, L'ISBN (*International Standard Book Number* ou numéro international normalisé du livre) a changé en 2007, passant de 10 à 13 caractères. Autre exemple : dans le monde du médicament vétérinaire, l'ancien code AMM a été remplacé il y a quelques années par un code plus long, le GTIN. Autre raison d'ailleurs pour choisir une clé technique : elle sera souvent plus compacte qu'un identifiant naturel.

Une des problématiques de la vie d'un développement consiste à gérer le changement des objets du domaine, donc des classes. Les classes sont liées entre elles par de nombreuses relations, ce qui complexifie le modèle et rend plus difficile son évolution. Eric Evans apporte une solution possible à ce problème : l'agrégation. Un agrégat *(aggregate)* représente un ensemble d'objets joints et considérés comme un tout du point de vue de la manipulation des données. Les objets sont regroupés à l'intérieur d'un objet racine *(root)* et en sont dépendants. Si une relation doit être effectuée entre des objets séparés, il faudra passer l'objet racine, qui lui seul porte une référence visible de l'extérieur. Cela permet non seulement une simplification du modèle, mais aussi une simplification des contraintes transactionnelles, du maintien de la cohérence et du verrouillage.

Eric Evans indique que si ces objets sont persistés en base de données, seul un accès par le root et son identifiant est possible. C'est donc exactement la façon dont nous pensons le stockage dans un moteur NoSQL orienté clé-valeur ou orienté documents comme MongoDB.

Les données dans une base NoSQL doivent donc être pensées en unités, comme un fichier, un document JSON, une valeur scalaire ou une ligne dans une famille de colonnes. Cette unité représentera ce que vous souhaitez écrire ou lire de façon cohérente et atomique, ce que vous voulez sharder, ce que vous allez identifier uniquement et requêter par une clé unique.

La cohérence

La question de la cohérence des données est également un point important. Dans un SGBDR, il est inutile de s'en préoccuper car c'est un fait acquis. Pour vous assurer de la cohérence entre plusieurs opérations, il suffit d'initier une transaction au niveau de votre code au moment du développement. Il est inutile d'évaluer cette question lors de la phase de conception. En revanche, dans des moteurs NoSQL comme MongoDB, la cohérence n'est assurée qu'au niveau du document. Si vous devez maintenir des niveaux de cohérence dans vos données, vous devez donc vérifier que tout ce qui doit être maintenu cohérent est stocké dans le même document.

Une entreprise qui fera le choix du NoSQL pourra difficilement se passer de bases relationnelles pour stocker ses données transactionnelles. Comment déterminer à quel endroit placer telle ou telle donnée ? Et comment faire coexister les deux environnements ? Le choix sera principalement guidé par le type des données et les besoins d'utilisation après coup de ces dernières. Des données typiquement front-office pourront être stockées dans un moteur NoSQL, pour favoriser l'affichage sur un site web, le maintien de données peu structurées ou encore pour profiter, avec des outils comme Redis, de capacités de rendu rapide de listes, de classement, etc. En revanche, il est conseillé de conserver les données de back-office dans une base de données relationnelle, afin de pouvoir traiter l'information avec toute la souplesse de recherche et de calcul que permet le langage SQL, pour des besoins d'analyse par exemple.

Lorsque les données sont structurées dans une base relationnelle, il devient beaucoup plus facile d'effectuer des calculs sur ces données, comme des agrégats. Imaginez que vous développiez un site de ventes privées. Les informations suivantes figurent dans votre catalogue : la description des produits, les différentes tailles, les images, des commentaires éventuels d'utilisateurs, etc. Tout ceci représente des données plutôt orientées documents, peu structurées, de taille très variable et qui sont toutes référencées sur la clé du produit. Un moteur NoSQL est donc un candidat idéal pour stocker ces informations. Ainsi, vous utiliserez, par exemple, MongoDB pour stocker le document entier, comportant la description, les détails techniques, les liens sur des images et une liste de commentaires clients. Ce document sera indexé sur la clé, qui correspond au SKU de l'article.

SKU

SKU signifie *Stock-Keeping Unit*, parfois traduit en français par Unité de gestion de stocks (UGS). Il désigne une référence élémentaire qui sert à la gestion précise des volumes en vente.

Lorsqu'une commande est réalisée sur le site, l'application web inscrit dans la base de données relationnelle les informations relatives à la vente (SKU, identification du client, etc.) et décrémente éventuellement le nombre d'unités disponibles dans le document MongoDB. L'inscription de la vente dans la base relationnelle permet de maintenir une cohérence transactionnelle sur ces opérations importantes, de traiter les étapes de la vente, d'effectuer des calculs statistiques pour les afficher sur des tableaux de bord, et de les exporter par la suite dans un système décisionnel.

Design patterns

Les moteurs NoSQL conduisent à de nouvelles pratiques et de nouvelles façons de concevoir les données en général. Plutôt que de parler de modélisation stricte, il vaut mieux aborder le sujet sous forme de design patterns. Au lieu d'être contraignants, ces derniers permettent de profiter de bonnes pratiques et de solutions éprouvées pour résoudre les problèmes de développement souvent rencontrés. En voici quelques-uns, adaptés au NoSQL.

La réduction dimensionnelle

Le principe de la réduction dimensionnelle *(dimensionality reduction)* est simple : certaines données sont trop complexes pour être effectivement manipulées. Il est alors souhaitable de limiter leurs attributs (leurs dimensions) et d'en extraire les plus importantes, pour ne pas tenir compte des autres. Il s'agit d'une façon de limiter la complexité, qui est utile dans des cas comme les besoins d'apprentissage machine ou de gestion de données géographiques, par exemple. MongoDB inclut un petit système d'information géographique (voir l'étude de cas du chapitre 16) qui réduit la complexité, en n'offrant que des points et pas de polygones, et en considérant la Terre comme une sphère parfaite au lieu d'utiliser un système de coordonnées ou de projection. MongoDB n'est pas un système d'information géographique complet, mais il permet une gestion simplifiée de points géographiques pour des besoins modestes.

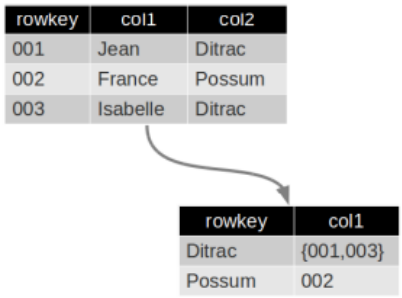
D'un point de vue pratique, cela signifie : identifier les besoins, éliminer les données inutiles (dont vous pouvez être sûr à 100 % qu'elles ne seront jamais utiles, méfiez-vous de ce point), ou diminuer la complexité par des modèles mathématiques, comme le calcul d'histogrammes statistiques, les méthodes de réduction ou la transformation de valeurs discrètes en valeurs continues.

La table d'index

Les index secondaires soulèvent de vraies questions dans le monde NoSQL, alors que leur utilisation ne fait aucun doute dans les SGBDR. Dans un moteur NoSQL, l'indexation secondaire est soit impossible, soit cantonnée à des recherches bien spécifiques. Une chose est certaine, peu d'index secondaires sont créés. Les données dans un moteur NoSQL ne sont pas aussi facilement manipulables que dans un moteur relationnel. Si vous souhaitez accéder à vos données par un autre critère que la clé, vous devez donc soit créer un index secondaire, soit générer vous-même une forme d'index au moyen de la technique de la table d'index, qu'on pourrait aussi appeler vue matérialisée. Nous avons illustré cette technique dans CouchDB, où elle est aisée grâce à l'utilisation de vues dans des documents de design, qui génèrent automatiquement des index. Cette technique est aussi couramment utilisée dans les moteurs basés sur Dynamo d'Amazon, comme Riak ou Cassandra, et dans Redis. On reprend simplement les données et on recrée une

collection dont la clé est la valeur à rechercher. La valeur sera une liste de références sur la clé de la table originelle. Il faut bien entendu s'assurer que cette deuxième table, ou collection, est toujours synchronisée avec la table originelle, ce qui se fait la plupart du temps dans le code client. La figure 14-1 illustre très simplement le concept.

Figure 14-1
Table d'index



Nous créons une nouvelle table et stockons en tant que valeur une liste de clés comportant le nom référencé par la clé. Une pratique utilisée dans Cassandra va d'ailleurs un peu plus loin. En effet, comme le nom des colonnes n'est pas fixé dans une famille de colonnes, qui est une *SortedMap* de colonnes, il est inutile de placer cette liste de référence dans la valeur de la colonne. Il suffit de nommer cette colonne avec la liste de référence et de n'y placer aucune valeur. Ceci est possible parce que le nom de la colonne ne doit pas obligatoirement être une chaîne, cela peut être n'importe quoi exprimé en octets.

La clé composite

Les types de données qui constituent la clé des moteurs NoSQL sont, dans la grande majorité des cas, très souples. Vous n'êtes pas limité à une chaîne de caractères alphanumériques stricte comme dans les SGBDR. Cela vous permet d'exprimer dans la clé, comme d'ailleurs dans le nom des colonnes dans une base orientée colonnes, plus d'informations qu'une simple valeur. Par exemple, vous pouvez ajouter l'équivalent d'espaces de noms en préfixe de vos clés ou de vos colonnes. Cette pratique est courante dans Redis ou Cassandra.

En Redis, comme nous travaillons souvent dans un espace de clés unique, il est utile de préfixer les clés pour indiquer à quoi elles servent. Prenons un exemple : nous voulons garder les informations d'un utilisateur, une liste de ses photos et une liste de ses amis. Nous pouvons structurer les clés comme suit :

```
User:rudi@babaluga.com
Images:rudi@babaluga.com
Friends:rudi@babaluga.com
```

La première clé indique que nous allons stocker les informations relatives à l'utilisateur dont l'identifiant est `rudi@babaluga.com`. La valeur sera probablement une table de hachage pour conserver

des paires clé-valeur de propriétés. Pour conserver les images, nous créons une liste d'adresses d'images sous forme d'URI, que nous stockons dans la clé `Images:rudi@babaluga.com`. Pour retrouver les images de l'utilisateur, notre code client n'aura qu'à créer la clé avec une concaténation de la chaîne «Images:» et de l'identifiant utilisateur, et à appeler un `LRANGE` de cette clé. Le raisonnement est le même pour la liste des amis de l'utilisateur.

Dans Cassandra, une pratique similaire permet de classer les colonnes présentes dans la famille de colonnes en sous-groupes, comme dans un schéma de la norme SQL. Dans une famille de colonnes `User`, nous pouvons créer des colonnes `Adresse:CodePostal`, `Adresse:Ville` et `Adresse:Pays`, ce qui permet de qualifier les colonnes, au lieu de créer des supercolonnes et de stocker l'adresse à l'intérieur (la supercolonne n'est pas toujours une bonne pratique en termes de performance).

Attention au langage CQL

Cette solution fonctionne très bien pour un accès natif à travers Thrift, mais attention si vous souhaitez utiliser le langage CQL. Celui-ci, pour conserver une forme de compatibilité avec SQL, ne reconnaît pas les noms de colonnes qui ne sont pas alphanumériques; Cela vous obligera à protéger vos colonnes par des guillemets doubles à chaque utilisation, ce qui peut se révéler pénible.

Les nested sets

Alors que les moteurs relationnels stockent des ensembles dans des tables séparées, les bases NoSQL ne gèrent pas de liens référentiels automatiques entre les données, même si rien ne les en empêche. MongoDB, par exemple, inclut un type `DBRef` qui représente la référence à une clé d'un autre document. Vous pouvez tout à fait créer un code JavaScript exécuté du côté serveur qui récupère cette référence et ajoute son contenu dans le document retourné. En revanche, il serait impossible par ce mécanisme d'obtenir les mêmes performances que les jointures d'un moteur relationnel, où les algorithmes de jointure sont optimisés. La pratique dans un moteur NoSQL sera alors d'inclure les sous-ensembles de données dans l'ensemble parent, par exemple sous la forme d'ensembles emboîtés *(nested sets)*. Prenons un exemple très simple de gestion de catégories de produits. Si nous voulons exprimer une hiérarchie de catégories de produits, nous pouvons le faire dans une collection MongoDB en hiérarchie pure, en indiquant par exemple un niveau et la référence du parent:

```
db.categories.insert ({_id: «vêtements», parent: null, niveau: 0})
db.categories.insert ({_id: «chaussures», parent: «vêtements», niveau: 1})
db.categories.insert ({_id: «jupes», parent: «vêtements», niveau: 1})
db.categories.insert ({_id: «salopettes», parent: «vêtements», niveau: 1})
db.categories.insert ({_id: «sous-vêtements», parent: «vêtements», niveau: 1})
db.categories.insert ({_id: «culottes», parent: «sous-vêtements», niveau: 2})
```

Mais nous pouvons également l'exprimer dans l'autre sens, en incluant une liste des sous-éléments, de la façon suivante:

```
db.categories.insert ({_id: «vêtements», descendants: [«chaussures», «jupes»,
«salopettes», «sous-vêtements»], niveau: 0})
db.categories.insert ({_id: «sous-vêtements», descendants: [«culottes», «caleçons»,
«soutien-gorge»], niveau: 1})
```

```
db.categories.insert ({_id: «caleçons», descendants: [«unis», «caleçons», «soutien-
gorge»], niveau: 1})
```

Ensuite, nous pouvons trouver la liste des catégories de niveau 1 (donc les descendants du niveau 0) :

```
db.categories.findOne ({ niveau: 0}).descendants
```

Ce qui retourne les descendants sous forme de tableau, facile à manipuler :

```
[«chaussures», «jupes», «salopettes», «sous-vêtements»]
```

Choisir l'architecture matérielle

À l'instar de la question de la modélisation des données, la problématique de l'architecture matérielle est moins importante en NoSQL que dans le domaine des moteurs relationnels. Un moteur relationnel, consigné sur une machine ou un petit nombre de nœuds, a des exigences plus fortes sur la qualité du matériel, le type de stockage à mettre en œuvre et les moyens d'assurer éventuellement la réplication de ce stockage dans un modèle de cluster. Dans le monde NoSQL, des moteurs comme Cassandra, Riak ou même MongoDB n'ont absolument pas ce type d'exigence. Au contraire, ils sont conçus pour être élégamment distribués sur des machines de puissance moyenne, remplaçables et interchangeables facilement, ce qu'on appelle du *commodity hardware*. Là où un moteur relationnel essaie d'optimiser son rapport au matériel, à travers des mécanismes internes de gestion des processeurs, de la mémoire et du disque, les moteurs NoSQL comptent plus sur une répartition du travail sur des nœuds qui individuellement n'ont pas besoin de dégager une forte puissance, mais qui font la force par leur union. Tout cela obéit à un principe d'économie propre aux réalités de l'informatique d'aujourd'hui : les machines de moyenne puissance coûtent peu cher et sont faciles à remplacer. Quelques points sont toutefois à respecter.

Tout d'abord, tous les moteurs peuvent profiter d'une bonne quantité de RAM. Cassandra et HBase conservent leurs données en mémoire dans un memstore, qui est régulièrement purgé sur le disque lorsqu'il est plein. Cassandra maintient en mémoire un cache de clés et un cache de données. Pour ces deux outils, il est utile de surveiller et de paramétrer la quantité de mémoire allouée pour obtenir les meilleures performances. En ce qui concerne Cassandra, la quantité de mémoire nécessaire est liée au nombre de familles de colonnes que vous définissez, puisque chaque famille de colonnes est maintenue dans un memstore.

Riak offre un choix de moteurs de stockage basé sur ce qui est disponible pour Erlang, mais globalement, l'architecture est semblable : WAL et écriture en RAM. MongoDB place ses données en cache en utilisant les fonctions de cache de fichiers fournis par le système (par fichiers projetés en mémoire à l'aide de la fonction `mmap`). Ceux-ci travaillent leurs données en RAM bien sûr, et profitent de grandes quantités de mémoire, spécialement lorsque le volume de données à utiliser est important.

CouchDB

CouchDB est une exception, il peut se contenter de peu de mémoire car il n'utilise aucune forme de cache en interne, autre que celle proposée nativement par le système d'exploitation. CouchDB est souvent utilisé dans des systèmes embarqués.

Il est donc important de soigner la quantité de mémoire et de choisir un système 64 bits, afin de ne pas souffrir de la limite des 4 Go imposée par le pointeur 32 bits. Il faut aussi surveiller le swap sur le disque. En effet, si un moteur NoSQL voit son espace de travail swappé à cause d'un manque de mémoire, les performances s'effondreront.

Il peut être utile également de tester les performances de votre système avec plus ou moins de RAM. Pour cela, vous pouvez utiliser un outil très simple, qui effectue une allocation de mémoire de la taille définie en paramètre (avec un simple appel à `malloc()`). Vous trouverez cet utilitaire sur le dépôt GitHub de MongoDB, à l'adresse suivante : https://github.com/mongodb/mongo-snippets/blob/master/cpp/eatmem.cpp. Un outil similaire pour Windows peut être trouvé dans cet article du MSDN Magazine : http://msdn.microsoft.com/en-us/magazine/cc163613.aspx. Cela vous permettra de mieux connaître vos besoins et de faire des tests de charge, ce qui est toujours très important. Nous en parlerons plus loin.

Évaluer les besoins en mémoire de Redis

Évidemment, la quantité de RAM est encore plus importante pour les bases de données en mémoire comme Redis, Membase ou Couchbase. Riak offre également un moteur de stockage en mémoire, mais qui devrait être réservé pour des tests.

Pour Riak, Basho propose un calculateur de quantité de RAM par rapport à la taille de vos données, si vous utilisez le système de stockage Bitcast (http://docs.basho.com/riak/1.1.0/references/appendices/Bitcask-Capacity-Planning/), principalement parce que le stockage Bitcast implique que toutes les clés soient maintenues en RAM. Nous reparlerons de Bitcast dans la section suivante dédiée au disque. Redis essaie d'être aussi efficient que possible dans sa consommation mémoire, mais il n'y a pas de miracle : toutes vos données doivent résider en RAM. À partir de la version 2.6, un algorithme d'optimisation de mémoire a été intégré au moteur, qui permet de diminuer de façon conséquente l'empreinte en RAM des données. Pour plus de renseignements, consultez la page web suivante : http://redis.io/topics/memory-optimization.

Mémoire virtuelle en Redis

À partir de la version 2, Redis a intégré un système nommé mémoire virtuelle *(virtual memory)* qui permet de gérer en interne un système de swap pour déplacer sur disque (idéalement un SSD) les valeurs les moins utilisées, toutes les clés restant quant à elles en mémoire. Ceci s'effectue grâce à un algorithme basé sur le temps de dernier accès et sur la taille des valeurs. Cette fonctionnalité a été dépréciée et retirée après la version 2.4.

Il n'est forcément aisé d'évaluer la mémoire nécessaire pour Redis. La méthode la plus simple consiste à partir de la taille actuelle des données et à calculer grossièrement les besoins en multipliant cette valeur avec le nombre d'éléments attendus. Pour afficher la taille actuelle des données, vous pouvez utiliser la commande `info` :

```
redis-cli info
```

qui retourne sur notre serveur d'exemple le résultat suivant, dont nous n'avons conservé que les informations relatives à la mémoire :

```
# Memory
used_memory:4330248
used_memory_human:4.13M
used_memory_rss:5623808
used_memory_peak:4292720
used_memory_peak_human:4.09M
used_memory_lua:20480
mem_fragmentation_ratio:1.30
mem_allocator:jemalloc-3.0.0
...
# Keyspace
db0:keys=15174,expires=0
```

Ici, nous avons une consommation mémoire de 4,13 Mo pour 15 174 clés, qui comportent des tweets. Il s'agit de l'application exemple que nous avons réalisée au chapitre 10. Les valeurs stockées ne sont pas très volumineuses. Cela nous donne une idée du nombre de valeurs que nous pouvons stocker avec notre quantité de mémoire. Cette occupation en RAM peut faire réfléchir sur la validité de l'utilisation de Redis pour de larges volumes. Ne désespérez tout de même pas trop vite. Il est possible de tricher un peu afin d'optimiser le stockage : profiter des tables de hachage, qui offrent un stockage optimisé. Au lieu de stocker vos clés indépendamment, vous pouvez regrouper des paquets de clés dans des tables de hachage dont les clés correspondent aux valeurs de rangée. Vous pourrez ainsi trouver rapidement la bonne clé pour ouvrir la bonne table de hachage qui contient la donnée que vous voulez utiliser. Théoriquement, vous pourriez même utiliser un algorithme basé sur du *consistent hashing* pour générer votre clé, ou essayer de travailler avec un filtre de Bloom. Cette méthode permet d'économiser pas mal d'espace. Vous trouverez une discussion sur cette technique sur le blog d'Instagram, qui l'utilise en production : http://instagram-engineering.tumblr.com/post/12202313862/storing-hundreds-of-millions-of-simple-key-value-pairs.

Attention aux sauvegardes sur disque

La mémoire occupée par Redis peut augmenter lors d'une opération de sauvegarde sur disque, par exemple en lançant la commande `BGSAVE`. Pour effectuer l'opération, Redis lance un sous-processus en effectuant un fork, qui va travailler sur un instantané *(snapshot)* des données. Les données entre les deux processus sont partagées, sauf celles qui sont écrites pendant l'existence du sous-processus : l'instantané devra se maintenir et ces données modifiées seront doublées en mémoire, l'ancienne version étant conservée pour le sous-processus. Sur un système qui génère beaucoup d'écritures, cela peut demander de la mémoire libre supplémentaire.

Pour connaître l'occupation mémoire actuelle, nous pouvons utiliser la commande `free` :

```
free -mt
```

ce qui retourne chez nous :

```
             total  used   free   shared  buffers   cached
Mem:  977     893     84        0       54       48
-/+ buffers/cache:        790        187
Swap:   999     93    906
Total: 1977   986    991
```

L'option `-m` demande un retour en Mo, `-t` affiche un total. Nous avons donc ici 977 Mo de RAM physique disponible. Nous ignorons les 999 Mo de swap, que nous ne voulons bien sûr pas utiliser.

Voyons maintenant, de l'extérieur, quelle est la consommation mémoire de Redis :

```
ps aux | grep [r]edis-server
```

ce qui retourne chez nous :

```
rudi      4825  0.2  0.5  31612  5492 pts/2    Sl+  15:46   0:33 redis-server
```

La commande ps

La commande `ps` retourne des informations sur les processus. Les options `aux` signifient, pour simplifier, « voir les détails de tous les processus ». Nous passons le résultat de la commande à l'outil `grep`, qui ne garde que les lignes qui nous intéressent. Nous commençons le modèle recherché par `grep` par des crochets (`[]`) autour de la première lettre, il s'agit d'un truc pour éviter de retourner la commande d'appel de `ps aux | grep` elle-même. Sans cela, nous aurions deux lignes : celle que nous cherchons, plus la commande que nous venons de lancer puisqu'elle contient la chaîne « redis-server ». Comment cela fonctionne-t-il ? `[]` indique un intervalle de caractères, nous cherchons donc un « r » suivi de « edis-server ». Nous ne trouvons pas la commande que nous lançons dans le résultat, puisque pour la trouver il faudrait maintenant chercher littéralement la chaîne « r]edis-server ».

Le résultat obtenu contient les colonnes suivantes :

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
```

Nous avons donc un pourcentage de mémoire utilisée, chez nous 0,5, puis une valeur pour VSZ (31612) et une valeur pour RSS (5492). Les résultats sont exprimés en octets. VSZ signifie *virtual set size* et RSS *resident set size*, soit la taille allouée par le système dans la mémoire virtuelle et la taille occupée physiquement dans la RAM. Cette valeur comporte les éventuelles bibliothèques partagées en mémoire avec d'autres applications, elle n'est donc pas parfaite. Vous pouvez afficher plus précisément la taille occupée en mémoire en utilisant la commande `pmap`. Voici un exemple d'appel pour Redis :

```
pmap -x $(pidof redis-server)
```

Nous récupérons le PID (identifiant de processus) de `redis-server` et nous l'utilisons pour `pmap`. L'option `-x` affiche un résultat étendu qui nous permet de voir le détail. Voici quelques lignes de résultat :

```
4825:   redis-server
Address   Kbytes     RSS   Dirty Mode   Mapping
```

```
08048000       0     392       0 r-x--  redis-server
080e7000       0       4       4 r----  redis-server
080e8000       0      12      12 rw---  redis-server
080eb000       0      40      40 rw---    [ anon ]
09e9a000       0      28      28 rw---    [ anon ]
b5400000       0     416     416 rw---    [ anon ]
b5bfe000       0       0       0 -----    [ anon ]
b5bff000       0       4       4 rw---    [ anon ]
b63ff000       0       0       0 -----    [ anon ]
b6400000       0    3872    3872 rw---    [ anon ]
b7601000       0       4       4 rw---    [ anon ]
b7602000       0     508       0 r-x--  libc-2.13.so
…
-------- ------- ------- ------- -------
total kB   31608       -       -       -
```

La valeur `Dirty` indique la valeur utilisée en mémoire physique, non swappée.

Appel avec sudo

Si le processus s'exécute avec un compte pour lequel vous n'avez pas de droit, vous devez lancer la commande `pmap` avec `sudo`.

Nous n'avons conservé que quelques lignes du résultat. Vous constatez que l'occupation est utilement séparée entre le processus lui-même, les bibliothèques partagées (qui apparaissent sur plusieurs lignes car la zone mémoire utilisée pour le code est séparée de la zone mémoire utilisée pour les données), des sections marquées `Anon`, qui indiquent des blocs de mémoire réservés par une fonction `malloc` ou `mmap`, et la pile (*stack*, l'endroit où son stockées les variables locales du code). Cette vision vous sera utile si vous voulez analyser plus précisément l'occupation mémoire de votre moteur.

Si vous voulez vraiment tous les détails, vous pouvez utiliser la vision donnée par le fichier virtuel `smaps` dans le répertoire `/proc` du processus. Nous avons vu précédemment que le PID de `redis-server` est 4825, nous pouvons donc inspecter ce fichier ainsi :

```
cat /proc/4825/smaps
```

La valeur `Private_Dirty` que nous trouverons dans ce fichier est l'ultime précision que nous pouvons obtenir sur l'utilisation réelle de la mémoire physique de chaque élément de notre programme.

Nous sommes entrés un peu dans les détails pour vous donner des outils génériques afin d'analyser la consommation mémoire de n'importe quel processus, donc moteur NoSQL. Chaque moteur peut toutefois publier ses propres informations, comme nous l'avons vu avec Redis.

Évaluer les besoins disque

L'évaluation des besoins disque n'est pas forcément aisée non plus. Les moteurs orientés documents peuvent consommer beaucoup plus d'espace disque que les moteurs relationnels, à cause de la redondance des données dans leur modèle.

CouchDB est particulièrement gourmand à cet égard, pour plusieurs raisons.

- Il maintient plusieurs versions du même document JSON. Il faut donc effectuer des compactages réguliers.
- La création de vues génère des index, donc des structures redondantes, qui sont persistées sur disque. Ces index doivent se maintenir, au détriment de l'espace disque. Les vues doivent aussi être compactées.
- Lorsque les vues sont modifiées, d'anciens index peuvent rester sur le disque. Une commande existe pour le nettoyage : `_view_cleanup`.

Les différentes options de compactage sont documentées à l'adresse suivante : http://wiki.apache.org/couchdb/Compaction.

MongoDB

En MongoDB, les fichiers sont préalloués avec des tailles fixes. Lorsque les fichiers atteignent une taille de 2 Go, les prochaines préallocations de fichiers s'effectuent par incréments de 2 Go. Un certain nombre de commandes permettent de déterminer la taille réelle des données. En voici un exemple pour notre collection `articles` :

```
db.articles.dataSize(); // taille des données
db.articles.totalIndexSize(); // taille des index
db.articles.storageSize(); // allocation, comporte l'espace inutilisé
db.articles.totalSize(); // données et index
```

Cassandra

Les moteurs orientés colonnes comme Cassandra utilisent un schéma de stockage proche de celui des moteurs relationnels, avec des écritures séquentielles dans un journal puis, régulièrement, une écriture séquentielle en masse dans un fichier immutable (chaque famille de colonnes est stockée dans un fichier distinct), en mode *append*. Ce mécanisme limite au maximum les besoins de lectures aléatoires qui, rappelons-le, sont les points faibles des disques magnétiques. Le temps moyen d'accès en lecture aléatoire sur un disque magnétique est de 10 ms au moins. Connaissant cela, les quelques règles de base pour organiser le sous-système disque de serveurs faisant tourner Cassandra sont assez évidentes.

- Privilégier les disques avec de bons rendements en lecture mais aussi en écriture.
- Le SSD peut être une solution, mais il faut qu'il ait un bon ratio écriture/lecture.
- Compte tenu des besoins de stockage, généralement élevés, des bases Cassandra (nous évoluons en général dans du Big Data), il est préférable de ne pas dépenser en SSD et de se contenter de disques de performances moyennes, mais de distribuer la charge sur plusieurs machines, ce pourquoi Cassandra est conçu en premier lieu.
- Séparer le disque de journal *(commitlog)* du disque contenant les données *(SSTable)*.
- Privilégier le RAID 10 au RAID 5, qui est lent en écriture à cause du calcul de parité.
- Formater les disques avec un système de fichiers moderne comme ext4 ou XFS. La limite de taille d'un fichier en ext3 est de 2 To alors que ext4 supporte des fichiers de 16 To. Les fichiers

de Cassandra peuvent atteindre de grandes tailles. De plus, les performances de suppression de fichiers de grande taille sont bien meilleures en ext4 qu'en ext3, ce qui est bénéfique lors des phases de compactage.

Il est à noter que les fichiers de données (SSTable) de Cassandra étant immutables, ils s'accumulent pour une même famille de colonnes. Régulièrement, Cassandra va les fusionner et les compacter (regrouper les fichiers, nettoyer les lignes marquées en *tombstone*, etc.). Ceci implique deux choses : un pic d'activité disque à ce moment et un besoin d'espace libre souvent conséquent. Pour éviter les problèmes, il peut être nécessaire de laisser 50% d'espace disque libre sur vos partitions contenant les SSTable de Cassandra.

Leveled compaction

Le mode de compactage utilisé par défaut dans Cassandra (nommé *tiered compaction*) peut nécessiter dans le pire des cas le double d'espace de stockage. Depuis Cassandra 1.0, vous pouvez configurer vos familles de colonnes pour utiliser un mode de compactage beaucoup moins gourmand (environ 10% d'espace libre nécessaire par rapport au volume des données). Ce mode de compactage est appelé *leveled compaction*. Il est documenté sur la page web suivante : http://www.datastax.com/dev/blog/when-to-use-leveled-compaction.

Riak

Riak, de son côté, peut utiliser différents moteurs de stockage qui sont en fait des API de sérialisation disponibles en Erlang. Le moteur le plus utilisé est Bitcast, qui est structuré autour d'une table de hachage journalisée et optimisée pour stocker les paires clé-valeur. Riak crée un répertoire Bitcast par *vnode*. Pour éviter des accès disque aléatoires, Riak écrit autant que possible ses fichiers en mode *append-only*, ce qui peut provoquer une augmentation de la taille des répertoires à cause de la présence de données dupliquées. Lorsque la limite d'espace disque définie est atteinte, Riak opère un compactage en écrivant un nouveau fichier de données. Vous pouvez paramétrer la taille maximale de vos fichiers, les périodes pendant lesquelles un compactage *(merge)* peut être effectué par Riak, et quelles sont les conditions de déclenchement de ce compactage. Les détails sont disponibles sur cette page : http://docs.basho.com/riak/latest/tutorials/choosing-a-backend/Bitcask/. En résumé, une fois de plus il faut préférer des disques de bonne taille, avec de bonnes performances en écriture. Évitez vraiment le RAID 5.

Virtualisation

Bien entendu, les moteurs NoSQL peuvent être installés dans des machines virtuelles, comme celles gérées par VMWare ESX, ou Xen. Une machine virtuelle offre moins de performances qu'une machine physique, et cela peut se ressentir plus spécialement dans les moteurs qui font une utilisation intensive du matériel. Un moteur, relationnel ou NoSQL, est souvent fortement sollicité. Si c'est le cas, essayez de le faire tourner sur une machine physique ou au moins soignez la configuration de votre machine virtuelle en ce qui concerne les accès disque et la gestion de la mémoire, qui sont les deux éléments importants dans un moteur de bases de données (tout ce qui touche aux entrées/sorties).

Sur VMWare ESX, deux éléments au moins sont à prendre en compte. Tout d'abord, VMWare offre une fonctionnalité de partage de la mémoire entre les VM, nommée *overcommit*. Elle

consiste à donner à une machine virtuelle de la mémoire normalement réservée pour une autre qui ne l'utilise pas. Cette fonctionnalité doit être utilisée avec prudence, certains utilisateurs ont en effet rencontré des problèmes avec des moteurs comme Cassandra ou MongoDB. De toute manière, un serveur de bases de données fait souvent un usage intensif de la mémoire pour son cache, dans ce cas, l'*overcommit* est moins intéressant.

Ensuite, surveillez bien les performances des disques et assurez-vous que la configuration est correcte. Par exemple, chez un de nos clients, nous avons expérimenté de très mauvaises performances sur un SAN à cause du paramètre d'accès multichemin *(multipathing)* défini dans VMWare. Celui-ci était configuré en Circulaire *(Round-Robin)*. Après l'avoir passé en fixe, le temps de copie d'un fichier de 800 Mo est passé de 3 min 40 s. a environ 10 s...

Pour ce qui est des serveurs qui ne travaillent pas beaucoup (par exemple, des serveurs de configuration d'une installation shardée de MongoDB), il est bien sûr intéressant de les virtualiser.

Mettre en place la solution et importer les données

Comment s'assurer d'une bonne installation ? Importer et convertir les données existantes pour les intégrer dans une base NoSQL ?

Déploiement distribué

Le déploiement est complètement différent selon que vous installez une base de données en mémoire comme Redis ou un moteur distribué comme Cassandra ou HBase. Il n'y a rien de particulier à dire sur le déploiement de solutions monoserveurs, nous allons donc aborder le déploiement distribué.

Si vous devez mettre en place quelques serveurs, une approche basée sur des scripts shell sur un système Unix, ou Powershell pour Windows peut être suffisante. Dans le monde Unix, utilisez des scripts shell déployés par `ssh`, qui permet non seulement d'ouvrir une session interactive sur une machine distante à travers une connexion sécurisée, mais aussi d'exécuter une commande directement en la mentionnant comme paramètre de l'appel à `ssh`. En voici un exemple :

```
ssh rudi@carthage.babaluga.com df -h
```

Ici, nous exécutons directement la commande `df -h`. Elle retourne l'espace libre sur les disques de la machine avec un affichage aisé à lire (grâce à l'option `-h`) sur le serveur `carthage.babaluga.com` en s'identifiant comme utilisateur `rudi`. Cette commande peut être simplifiée encore en utilisant un alias ou un script shell. En utilisant le shell bash, nous pouvons ajouter cette ligne dans notre fichier `~/.bashrc` :

```
alias carthage='ssh rudi@carthage.babaluga.com'
```

Il nous suffira alors de lancer notre commande ainsi :

```
carthage df -h
```

pour obtenir le même résultat. Il est ensuite aisé de boucler sur plusieurs serveurs, comme dans l'exemple suivant :

```
for i in 192.168.0.22 192.168.0.100; do
       ssh rudi@$i df -h
done
```

qui lance la commande `df -h` sur les machines accessibles sur les adresses IP 192.168.0.22 et 192.168.0.100 avec le compte `rudi`.

Authentification par clé

Évidemment, pour rendre cette méthode utile en pratique, nous avons dû au préalable configurer une sécurité basée sur une paire de clés. Dans le cas contraire, il aurait fallu entrer notre mot de passe (celui du compte `rudi`) à chaque tentative de connexion. La configuration d'une authentification `ssh` par clé dépasse le cadre de cet ouvrage. Pour une description claire de la marche à suivre, consultez la page suivante: http://doc.ubuntu-fr.org/ssh#authentification_par_un_systeme_de_cles_publiqueprivee.

Notons que l'un des problèmes de cette approche apparaît avec des commandes plus complexes: la gestion des apostrophes pour protéger une chaîne peut s'avérer cauchemardesque et elle est assez mal supportée selon les systèmes.

De la même façon, nous pouvons copier des fichiers sur les différents serveurs à l'aide de `scp`.

```
scp ~/cassandra/cassandra.yaml rudi@carthage.babaluga.com:/etc/cassandra/cassandra.yaml
```

La commande `scp` copie le fichier de configuration `cassandra.yaml` situé sur la machine locale vers le répertoire distant `/etc/cassandra/`. Cette approche est elle aussi assez frustre, elle ne fonctionne que par remplacement de tout le fichier. Bien entendu, des commandes `sed` peuvent être envoyées vers les machines distantes pour effectuer les remplacements, et c'est parfaitement envisageable dans de petits environnements. Il y a aussi quelques initiatives qui permettent de pousser un peu plus l'utilisation de `ssh`. Ainsi, les outils parallel-ssh (http://code.google.com/p/parallel-ssh/) en Python et ClusterSSH (http://sourceforge.net/apps/mediawiki/clusterssh/) en Perl peuvent être utilisés en tant que bibliothèques dans leur langage respectif pour créer des scripts plus complexes. Citons également PyDSH (http://pydsh.sourceforge.net/index.php), un shell distant écrit en Python. Il est aussi possible d'utiliser un gestionnaire de codes sources distribué comme git, ce qui présente l'avantage de maintenir aussi les versions des fichiers de configuration. On peut tirer régulièrement les modifications depuis les machines en planifiant une commande `git pull` via cron.

Des outils de gestion de configuration distribuée libres ont fait leurs preuves et sont quasiment indispensables lorsque le parc de machines atteint un certain seuil. Les deux outils les plus utilisés sont CFEngine et Puppet.

Outils de gestion de configuration

CFEngine et Puppet, bien que répondant à des besoins similaires, sont des outils assez différents. CFEngine est développé en C et interprète un langage déclaratif qui décrit l'état du système à installer. Sa conception est basée sur la théorie des promesses *(promise theory)*, c'est-à-dire un

modèle multi-agent où chacun publie ses intentions sous forme de promesses. Ceci constitue une base théorique à la gestion basée sur les stratégies *(Policy Based Management)*, mais en instillant l'idée d'agents autonomes plutôt que de machines forcées par des obligations. À partir de ce principe, chaque machine conserve une certaine autonomie. Un serveur centralisé stocke les stratégies, qui sont distribuées sur les hôtes, ce qui permet à CFEngine d'appliquer les règles sans se référer au serveur maître. CFEngine assure également une récupération des erreurs et des changements de configuration. En d'autres termes, si une mise à jour du système ou des changements manuels troublent la configuration telle que décrite dans CFEngine, celui-ci sera capable de corriger la différence et de revenir à l'état souhaité. De plus, le modèle de CFEngine est relativement décentralisé : chaque machine exécute un démon qui agit indépendamment des autres. De ce point de vue, CFEngine est basé sur un modèle similaire aux moteurs NoSQL décentralisés comme Cassandra ou Riak.

Sa prise en main est moins aisée que Puppet, mais la phase d'apprentissage en vaut la peine. Comme sa mise en œuvre est complexe, nous n'allons pas en parler dans ce livre, il faudrait lui consacrer un chapitre à part entière. Sachez simplement qu'une interface web pour CFEngine nommée Rudder est développée par Normation (http://www.normation.com/). Par ailleurs, si vous vous décidez à utiliser CFEngine sérieusement pour votre administration, nous vous recommandons la lecture du livre *Automating Linux and UNIX System Administration*, de Nathan Campi et Kirk Bauer, paru aux éditions Apress.

Importer les données

Chaque moteur comporte ses propres outils pour importer et parfois exporter des fichiers plats, comme du CSV (données séparées par des virgules), afin de réaliser des transferts de données en masse. La plupart du temps, vous n'aurez pas à développer une routine vous-même. CouchDB représente une exception car il ne propose pas d'outil d'import-export. Pour importer vos données, vous devrez écrire du code. Pour ce qui est de l'exportation, vous pourrez créer ce que CouchDB appelle une fonction de liste, qui permet de montrer les données dans le format que vous souhaitez. Pour plus d'informations, consultez cette page de documentation : http://guide.couchdb.org/editions/1/en/transforming.html.

MongoDB

MongoDB propose deux outils en ligne de commande pour effectuer des imports et des exports de fichiers au format JSON ou CSV. Notez que le format de stockage MongoDB (BSON) comporte des types légèrement différents du JSON originel ; vous pouvez donc rencontrer des différences entre le JSON généré et les données stockées. La liste des différences est disponible à l'adresse suivante : http://docs.mongodb.org/manual/administration/import-export/.

Voici quelques exemples d'exports :

```
mongoexport --db passerelles --collection articles --out ~/passerelles.articles.json
mongoexport --db passerelles --collection articles --jsonArray --out ~/passerelles.
articles.json
mongoexport --db passerelles --collection articles --query '{ "auteur.nom":
"Brizard"}'  --out   ~/passerelles.articles.brizard.json
```

```
mongoexport --db passerelles --collection articles --csv --out ~/passerelles.articles.
csv --fields=auteur.nom,auteur.e-mail
```

La première commande exporte la collection `articles` de la base `passerelles` dans un seul fichier JSON. La deuxième fait de même mais génère un fichier contenant un tableau de documents JSON. La troisième filtre les données pour n'exporter que les documents où le nom de l'auteur est `Brizard`. Enfin, la dernière commande exporte le nom et l'e-mail de l'auteur dans un fichier séparé par des virgules, dont nous reproduisons le contenu ci-après.

```
auteur.nom,auteur.e-mail
"Brizard","annie.brizard@cocomail.com"
"Brizard","annie.brizard@cocomail.com"
```

Vous noterez qu'une ligne d'en-tête avec le nom des colonnes a été ajoutée par `mongoexport`. Voici un exemple d'import de ce fichier CSV dans une nouvelle collection `auteurs`, qui sera automatiquement créée à l'import :

```
mongoimport --db passerelles --collection auteurs --type csv --file ~/passerelles.
articles.csv -headerline
```

Une notification nous indique que trois lignes ont été importées. La ligne d'en-tête est comptée dans l'import, mais ne vous inquiétez pas, grâce à l'option `-headerline`, elle ne sera pas considérée comme une ligne de données. Voyons le contenu de notre collection dans l'invite mongo :

```
> use passerelles;
switched to db passerelles
> db.auteurs.find();
{"_id": ObjectId("50e2e278b95794cfad02012e"), "auteur.nom": "Brizard", "auteur.e-
mail": "annie.brizard@cocomail.com"} {"_id": ObjectId("50e2e278b95794cfad02012f"),
"auteur.nom": "Brizard", "auteur.e-mail": "annie.brizard@cocomail.com"}
```

Comme vous pouvez le constater, à partir d'un import CSV, MongoDB crée une structure BSON plate, non hiérarchique : la liste `auteurs` n'a pas été recréée.

Riak

L'import et l'export « natifs » en Riak s'appuient sur des fonctions Erlang qui ont été implémentées dans les modules `bucket_importer` et `bucket_exporter`. Pour consulter le code source de ces fonctions, consultez les adresses suivantes : http://contrib.basho.com/bucket_importer.html et http://contrib.basho.com/bucket_exporter.html. Les modules développés permettent de définir un répertoire contenant des fichiers JSON pour les importer dans Riak. Pour des besoins plus précis, vous devrez développer les routines vous-même.

Redis

Redis ne comporte pas d'outil interne pour l'importation de fichiers CSV. Plusieurs techniques peuvent être déployées pour effectuer une insertion en masse. L'important est d'éviter d'effectuer une importation en boucle avec des instructions uniques, surtout si vous accédez au serveur Redis à travers le réseau. Désormais, Redis permet le *pipelining* de commandes, c'est-à-dire l'envoi

d'un batch de commandes en un seul appel, soit avec les commandes `MULTI` / `EXEC`, soit avec la fonctionnalité de pipelining de votre bibliothèque cliente. Voici un extrait de code Python utilisant le client Redis :

```
pipe = r.pipeline(transaction=False)
pipe.sadd(cle, valeur)
…
pipe.execute()
```

Nous ne développerons pas davantage cet exemple, car nous utiliserons ce code dans l'étude de cas du chapitre 16. Reportez-vous à ce chapitre pour un exemple complet.

En ligne de commande, vous pouvez utiliser le client Redis avec l'option `--pipe` qui définit un mode *pipe* relativement nouveau en Redis, disponible depuis la version 2.4. L'idée est de recevoir dans un pipe une liste de commandes Redis à exécuter en batch. Ces commandes doivent être écrites dans un format spécial, le protocole Redis, et non pas en texte simple, sinon le mode pipe ne pourra pas reconnaître les commandes. Ce protocole est simplement constitué de la longueur de la chaîne suivie de la chaîne elle-même après un CRLF, avec quelques caractères supplémentaires. La page de documentation de Redis (http://redis.io/topics/mass-insert) explique le protocole et offre une fonction Ruby pour générer les commandes dans ce protocole. Il est aisé de porter la fonction en Python, ce que nous avons fait en l'encapsulant dans un code qui prend en entrée un fichier contenant les commandes au format texte et qui retourne ces commandes dans le protocole Redis pour être utilisées directement en pipe avec `redis-cli`. Voici le code Python du fichier que nous avons appelé `redis-import.py` :

```
#!/usr/bin/python
import sys, re

def redis_proto(ligne):
    mots = ligne.split()
    proto = '*' + str(len(mots)) + '\r\n'
    for mot in mots:
        proto += '$' + str(len(mot)) + '\r\n' + str(mot) + '\r\n'
    return proto

with open(sys.argv[1], 'r') as fichier:
    for ligne in fichier:
        sys.stdout.write(redis_proto(ligne))
```

Créons maintenant un fichier texte contenant des commandes Redis, nommé `import-redis.txt`. Il est composé des lignes suivantes :

```
SELECT 1
SET 001 caillou
SET 002 hibou
SET 003 chou
SET 004 genou
```

Si nous appelons notre script avec notre fichier en entrée, comme ceci :

```
python redis-import.py import-redis.txt
```

voici le début de ce qu'il nous retourne sur la sortie standard (`STDOUT`) :

```
*2
$6
SELECT
$1
1
*3
$3
SET
$3
001
$7
caillou
…
```

Ensuite, exécutons l'import en passant ce résultat à `redis-cli` :

```
python redis-import.py import-redis.txt | redis-cli --pipe
```

Redis nous répond :

```
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 5
```

Nos données sont bien intégrées dans la base de données `1`, nous pouvons le vérifier dans `redis-cli` :

```
>select 1
OK
[1]> dbsize
(integer) 4
[1]> GET 002
"hibou"
```

Importer nos données dans Redis nécessite donc un peu de programmation, mais c'est assez aisé et efficace.

Cassandra

Cassandra propose plusieurs outils pour réaliser des imports et des exports. Le plus simple consiste à faire appel à des commandes CQL créées sur le modèle de commandes PostgreSQL : `COPY FROM` et `COPY TO`. La commande suivante, saisie dans une session `cqlsh`, exporte le contenu de la famille de colonnes `articles` dans un fichier `articles.txt` situé dans le répertoire personnel de l'utilisateur, en séparant les colonnes par une barre verticale, et en ajoutant en première ligne le nom des colonnes :

```
COPY articles (KEY, auteur_email, auteur_nom, auteur_prenom, titre) TO '~/articles.
txt' WITH DELIMITER = '|' AND QUOTE = ''' AND ESCAPE = ''' AND NULL = '<null>' AND
HEADER = 'true';
```

La commande suivante réimporte le même fichier dans une famille de colonnes nommée `articles_import`, que nous devons créer au préalable :

```
create table articles_import (KEY varchar PRIMARY KEY);
COPY articles_import (KEY, auteur_email, auteur_nom, auteur_prenom, titre) FROM '~/
articles.txt' WITH DELIMITER = '|' AND QUOTE = ''' AND ESCAPE = ''' AND HEADER =
'true';
```

NULL

Notez que l'option `NULL` n'est pas reconnue dans le `COPY FROM`. L'inconvénient de cette méthode est qu'en présence de `<null>`, la colonne se créera quand même et stockera la valeur de chaîne « <null> ».

Vous pouvez obtenir de l'aide sur ces commandes en tapant ceci dans une session `cqlsh` :

```
help copy;
```

Les utilitaires `sstable2json` et `json2sstable` permettent d'exporter une famille de colonnes contenue dans une SSTable vers un document JSON, et inversement. Son appel est très simple.

Un outil nommé `sstableloader` vous permet également de réaliser des imports et des exports à partir de SStable. Il nécessite un peu de développement.

Avec du code utilisant Python

Voyons maintenant une méthode générique implémentée dans du code Python. Pour exemple, nous allons insérer le contenu de ce petit fichier CSV, nommé `import-redis.csv` :

```
userid;nom;prenom;email
001;bruchez;rudi;rudi@babaluga.com
002;dupuis;sidonie;sidonie@aglae.com
```

dans une base Redis, à l'aide de ce script Python, que nous sauvegardons dans le fichier `import-csv.py` :

```
#!/bin/python
import csv, redis

r = redis.StrictRedis(host='localhost', port=6379, db=2)

with open("import-redis.csv", "rb") as f:
    for ligne in csv.DictReader(f, delimiter=';', quoting = csv.QUOTE_NONE):
    r.hmset("user:%s"% ligne["userid"], {"nom":ligne["nom"],
"prenom":ligne["prenom"], "email":ligne["email"]})
```

Ici, nous ouvrons une connexion à la base 2 de notre serveur Redis, puis nous utilisons le module `csv` de Python pour stocker le contenu du fichier CSV dans un dictionnaire. Nous insérons ensuite chaque ligne dans une table de hachage Redis à l'aide de la commande `hmset`, qui nous permet de générer la table de hachage en un seul appel. Ce code est fourni comme exemple simple, il n'est certainement pas très rapide. Par exemple, il vaudra mieux en cas de volume important utiliser du pipelining et récupérer les lignes de `csv` en utilisant la fonction `csv.reader()` qui stocke les colonnes dans une liste au lieu de créer un dictionnaire. Vous pouvez trouver un didacticiel très clair sur le module `csv` de Python à cette adresse : http://www.chicoree.fr/w/Fichiers_CSV_en_Python. Notez également que si vous développez en Ruby, le module `FasterCSV` est à préférer au module `csv` de la bibliothèque standard.

Nous exécutons notre code :

```
python import-csv.py
```

Il nous reste à vérifier la présence de nos deux lignes avec `redis-cli` :

```
redis 127.0.0.1:6379> select 2
OK
redis 127.0.0.1:6379[2]> dbsize
(integer) 2
redis 127.0.0.1:6379[2]> hmget user:001
(error) ERR wrong number of arguments for 'hmget' command
redis 127.0.0.1:6379[2]> hgetall user:001
1) "nom"
2) "bruchez"
3) "prenom"
4) "rudi"
5) "email"
6) "rudi@babaluga.com"
```

Importer du XML

Généralement, les moteurs NoSQL ne comportent pas d'outil intégré qui permet les échanges de données, comme les outils ETL *(Extract Tranform Load)* du monde décisionnel. Comme vous l'avez déjà compris, il faut écrire du code. C'est aussi le cas pour importer des documents XML dans les moteurs orientés documents. Grâce aux bibliothèques présentes dans les langages pour traiter facilement du XML, ce sera une tâche aisée. Nous prenons ici un court exemple avec Python et MongoDB. Python est une fois de plus un excellent langage pour ce genre de choses, car un grand nombre de bibliothèques existent pour manipuler des structures de données dans l'esprit de Python. Nous allons utiliser ici le module `xmltodict` qui, comme son nom l'indique, permet de transformer un document XML en dictionnaire Python. Cela tombe bien, puisque les méthodes d'insertion de documents du pilote `pymongo` utilisent des dictionnaires Python pour les transformer en interne en BSON. L'opération devient alors très simple : il nous suffit de passer le XML dans notre module et d'envoyer le résultat à `pymongo`. Tout d'abord, nous installons `xmltodict` :

```
~/python_env/nosql/bin/pip install xmltodict
```

Le code de notre script est ensuite le suivant. L'idée étant de récupérer un document XML, nous avons choisi le flux RSS du site d'information de France Info, dont l'adresse est http://www.franceinfo.fr/rss/tag/1.

```
#!/usr/bin/python
# coding=utf-8

import urllib
import xmltodict
import pymongo
from pymongo import Connection
from pymongo.errors import ConnectionFailure

rss = 'http://www.franceinfo.fr/rss/tag/1'

document = urllib.urlopen(rss)
o = xmltodict.parse(document)

try:
        cn = Connection(host="localhost", port=27017)
        print "connexion réussie"
except ConnectionFailure, e:
        sys.stderr.write("Erreur de connexion à MongoDB:%s"% e)
        sys.exit(1)

db = cn["france_info"]
assert db.connection == cn
print "handle acquis avec succès"

db.actus.insert(o, safe=True)
```

Nous utilisons d'abord le module `urllib` pour accéder au lien et récupérer le document, que nous transformons en dictionnaire à l'aide de `xmltodict.parse()`. Nous initions ensuite une connexion à MongoDB et nous insérons simplement ce dictionnaire dans une collection nommée `actus` de la base de données `france_info`. Voilà, rien de plus simple. Si nous enlevions la gestion des erreurs et si nous simplifiions le code, il pourrait tenir en quelques lignes.

Il nous reste à constater, dans le shell de MongoDB, que les données sont bien insérées dans la collection :

```
use france_info
db.actus.find()
```

Voici un extrait formaté du résultat :

```
{
    "title": "Paséo: famille, je vous \"aide\"...",
    "link": "http://www.franceinfo.fr/actu/initiative-france-info/
paseo-famille-je-vous-aide-831091-2012-12-13",
```

```
    "description": "Pour alléger le quotidien parfois pesant des familles
modestes, à Nîmes, l'association Paséo reprend un concept vieux d'une
quarantaine d'années en Grande Bretagne. Des parents bénévoles viennent
donner un coup de main une fois par semaine gratuitement aux foyers qui en
ont le plus besoin.",
    "category": [{"@domain": "http://www.franceinfo.fr/actu", "#text":
"Actu"},
     {"@domain": "http://www.franceinfo.fr/economie", "#text": "Économie"}],
    "pubDate": "Thu, 13 Dec 2012 06:20:00 +0100",
    "dc:creator": "Lucie Montchovi"
}
```

Un autre exemple d'import de XML dans MongoDB en utilisant C# est disponible à cette adresse : http://loosexaml.wordpress.com/2011/12/29/loading-xml-into-mongodb/.

Exemples de développements

Pour vous aider à développer avec les moteurs NoSQL, de nombreuses applications de démonstration en source libre sont à votre disposition. N'hésitez pas à vous plonger dans leur code pour vous inspirer. Voici quelques exemples :

- La distribution de Cassandra par DataStax comporte une application d'exemple en Java, nommée Portfolio Manager, qui utilise le pilote JDBC du langage CQL. Vous en trouvez un descriptif à cette adresse : http://www.datastax.com/docs/1.0/getting_started/dsc_demo ;
- Twissandra est une application similaire à Twitter, utilisant Cassandra comme moteur de bases de données. Vous pouvez la tester sur http://twissandra.com/. Malgré son interface un peu spartiate, les fonctionnalités sont basiquement semblables à Twitter. Les codes sources de cette application sont disponibles à l'adresse suivante : https://github.com/twissandra/twissandra ;
- Des programmes de démonstration semblables existent pour Redis, disponibles à l'adresse suivante : http://redis.io/topics/twitter-clone. Vous y trouverez notamment l'application originale en PHP (retwis), ainsi qu'une implémentation en Ruby avec le framework Sinatra et en Java avec Spring. Notez que les commandes de Redis peuvent évoluer au fil des versions, et que le code d'exemple peut ne pas toujours refléter la manière la plus efficace de manipuler les données Redis à l'heure actuelle ;
- Un autre exemple intéressant d'application basée sur Redis est haplocheirus (https://github.com/twitter/haplocheirus), un système de gestion de timelines (lignes de temps). L'un des intérêts de ce projet est qu'il ajoute une couche de sharding de la base Redis sur plusieurs nœuds en utilisant `gizzard` (`https://github.com/twitter/gizzard`), une bibliothèque qui répartit les données entre le client et le serveur ;
- NodeCellar (http://nodecellar.coenraets.org/) est une application d'exemple utilisant NodeJS avec MongoDB comme moteur de données. Comme son nom l'indique, il s'agit d'une application de gestion de caves à vin. Le code source est disponible sur GitHub : https://github.com/ccoenraets/nodecellar ;

- En ce qui concerne Riak, une liste de petites applications d'exemples est disponible sur le site de Basho : http://docs.basho.com/riak/1.3.0/references/appendices/community/Sample-Applications/ ;
- Enfin, nous avons déjà mentionné l'application Sofa, développée pour le livre *CouchDB: The Definitive Guide*. Il s'agit d'une application de blog totalement développée en CouchDB en utilisant CouchApp. Son code source est disponible à l'adresse suivante : https://github.com/jchris/sofa.

15

Maintenir et superviser ses bases NoSQL

Les moteurs NoSQL sont, au même titre que les moteurs relationnels, des applications serveur. Ils répondent de manière centralisée à des requêtes clientes, et sont de ce fait des applications centrales et décisives pour le bon fonctionnement des applications. Il est important d'assurer une supervision sérieuse des serveurs et de leurs performances. Nous allons voir dans ce chapitre quelques éléments de supervision et d'analyse propres aux moteurs NoSQL.

Réaliser des tests de charge

Dans le domaine des applications utilisant une base de données, il est très important de réaliser des tests de charge de manière à savoir quel est le nombre d'utilisateurs et le volume de données qu'est capable de supporter un serveur. Traditionnellement, les applications utilisant des moteurs relationnels sont développées avec des bases de données de test qui sont peu volumineuses. Bien entendu, l'application est développée dans un contexte peu concurrentiel, où seuls quelques développeurs accèdent à la base. Dans ces conditions, il est impossible de détecter les points de ralentissement durant la phase de conception. La volumétrie évoluant et le nombre d'accès simultanés se multipliant, les gros problèmes de performances apparaissent au plus mauvais moment, lorsque la solution est en production et durant les pics d'activité où l'entreprise fonctionne à plein régime. La catastrophe n'est pas loin.

Théoriquement, les moteurs NoSQL sont plus souples à ce niveau : HBase, Cassandra, Riak et MongoDB sont capables de monter en charge horizontalement et automatiquement par l'ajout de machines dans le cluster. Pour supporter un pic d'activité, il suffit donc d'ajouter de nouvelles machines. Dans la pratique, il est utile et important de réaliser des tests de charge pour savoir combien d'utilisateurs un nœud est capable de supporter et ainsi pouvoir planifier une capacité par machine, et donc la taille du cluster. De plus, dans le cas de moteurs comme Redis, où le but est d'offrir de la rapidité sur une seule machine, la problématique se pose comme dans le cas des moteurs relationnels.

Les tests de charge comportent deux aspects : ajouter des données en masse pour augmenter la volumétrie, et simuler un grand nombre d'utilisateurs pour tester les capacités de concurrence. Le premier aspect n'est pas compliqué, il faut développer spécifiquement du code qui va alimenter l'entrepôt avec des données de test, à l'aide de boucles. L'exemple du chapitre 10 qui consistait à récupérer des données de Twitter pour les insérer dans Redis est un exemple de code permettant de générer du volume.

Les imports et les exports

N'oubliez pas de tester votre système NoSQL avec des imports ou des exports de données si vous devez en faire, car ce sont des opérations qui peuvent consommer beaucoup de RAM.

Redis

Certains moteurs livrent des outils de benchmarking, c'est notamment le cas de Redis. L'outil s'appelle `redis-benchmark`, il permet de simuler des `GET` et des `SET` envoyés par un certain nombre d'utilisateurs simultanés. Voici un exemple d'appel :

```
redis-benchmark -q -n 100000 -d 2000 -k 0
```

Les options de la commande sont les suivantes :

- `-q` *(quiet)* : n'affiche que les informations importantes ;
- `-n 100000` : nombre de requêtes à envoyer ;
- `-d 2000` : taille des données envoyées par les `GET` et `SET` en octets, par défaut 2 octets ;
- `-k 0` : force les reconnexions, sinon la connexion reste en *keep alive*.

La commande peut échouer si le système d'exploitation a désactivé la fonctionnalité de *keep alive*. Dans ce cas, la pile réseau ne pourra pas créer le nombre de connexions demandées avec assez de célérité. `redis-benchmark` affichera probablement ce message :

```
WARNING: keepalive disabled, you probably need 'echo 1 > /proc/sys/net/ipv4/tcp_tw_
reuse' for Linux and 'sudo sysctl -w net.inet.tcp.msl=1000' for Mac OS X in order to
use a lot of clients/requests.
```

Sur Linux, nous avons donc activé la fonctionnalité `tcp_tw_reuse` à l'aide de la commande suivante, exécutée en tant que root.

```
echo 1 > /proc/sys/net/ipv4/tcp_tw_reuse
```

puis nous avons lancé la commande. Après un certain temps d'exécution, voici le résultat final :

```
PING_INLINE: 7962.42 requests per second
PING_BULK: 7968.76 requests per second
SET: 6529.55 requests per second
GET: 6578.08 requests per second
INCR: 7908.89 requests per second
LPUSH: 5566.69 requests per second
LPOP: 5339.88 requests per second
SADD: 7158.71 requests per second
SPOP: 7866.58 requests per second
LPUSH (needed to benchmark LRANGE): 6118.45 requests per second
LRANGE_100 (first 100 elements): 1565.14 requests per second
LRANGE_300 (first 300 elements): 568.47 requests per second
LRANGE_500 (first 450 elements): 378.38 requests per second
LRANGE_600 (first 600 elements): 282.54 requests per second
MSET (10 keys): 4092.32 requests per second
```

Nous obtenons un nombre d'opérations par seconde, qui correspond ici à notre machine virtuelle avec un processeur cadencé à 3,1 GHZ et avec 1 Go de RAM.

Cassandra

Cassandra est également livré avec un outil de test de charge, écrit en Java et nommé `cassandra-stress`. La liste des options est disponible dans la documentation à l'adresse suivante : http://www.datastax.com/docs/1.1/references/stress_java. Il y a quelques options intéressantes, notamment :

- `--enable-cql` : permet de tester les requêtes en CQL plutôt qu'en langage natif, on peut ainsi voir les différences de performance ;
- `--nodes` : liste de nœuds sur lesquels il est possible d'effectuer le test ;
- `--threads` : nombre de threads à exécuter, 50 par défaut.

Sur notre système, nous avons lancé `cassandra-stress` avec les options par défaut. Voici quelques lignes du résultat :

```
Created keyspaces. Sleeping 1s for propagation.
total,interval_op_rate,interval_key_rate,avg_latency,elapsed_time
8124,812,812,0.05000036927621861,10
20162,1203,1203,0.036634906130586474,20
35208,1504,1504,0.0280090389472285,30
54869,1966,1966,0.01964879711103199,40
73493,1862,1862,0.019572594501718214,50
93131,1963,1963,0.01855698136266422,61
115534,2240,2240,0.014769673704414588,71
146544,3101,3101,0.014638471460819091,81
180520,3397,3397,0.014889157052036733,91
216254,3573,3573,0.014129176694464656,101
250427,3417,3417,0.014769057443010564,111
285664,3523,3523,0.014240514232199108,121
320086,3442,3442,0.014762826099587473,132
```

```
358070,3798,3798,0.013284830454928391,142
395201,3713,3713,0.013596321133284857,152
```

Le résultat est retourné sous forme de tableau. La première valeur indique le nombre de lignes insérées. Par défaut, `cassandra-stress` effectuera l'insertion d'un million de lignes avant de s'arrêter. Les deuxième et troisième colonnes indiquent le taux d'opérations et de créations de clés, la quatrième colonne indique la latence moyenne des opérations, et la dernière valeur représente le nombre de secondes d'exécution de la rangée d'opérations.

Tests génériques

Pour les moteurs NoSQL qui ne possèdent pas d'outil de test de charge, vous pouvez écrire du code qui attaque le serveur avec des requêtes en créant plusieurs threads. Voici un exemple de code Python, qui utilise le module `threading` de Python, et qui reprend le code d'importation de documents XML dans MongoDB du chapitre 14 :

```
#!/usr/bin/python
# coding=utf-8

import threading
import time
import urllib
import xmltodict
import pymongo
from pymongo import Connection

rss = 'http://www.franceinfo.fr/rss/tag/1'
nombre_de_threads = 20

document = urllib.urlopen(rss)
o = xmltodict.parse(document)
nb_iterations = 20

def insert_mongo(iterations):
        cn = Connection(host="localhost", port=27017)

        db = cn["france_info"]

        count = 0
    while count < iterations:
            db.actus.drop()
            db.actus.save(o, safe=True)
            time.sleep(1)
         count += 1

# création de threads
count = 0
while count < nombre_de_threads:
        print "création du thread%s"% count
```

```
        threading.Thread(None, insert_mongo, None, (nb_iterations,)).start()
        count += 1
```

Ce code n'est pas parfait, notamment en ce qui concerne la gestion des threads qui est trop élémentaire et il sera difficile d'interrompre l'exécution de code immédiatement (pour une meilleure gestion des threads en Python, consultez la page suivante : http://python.developpez.com/faq/?page=Thread). Il s'agit d'un code de démonstration et l'essentiel y est. Nous déclarons une fonction qui va, à l'intérieur d'une boucle, supprimer la collection `actus` et la recréer en y insérant un document récupéré du flux RSS de France Info. Cette opération sera réalisée 20 fois par thread, avec une attente de 1 seconde entre chaque opération. La fonction sera exécutée en multithread, par 20 threads au total que nous créons et démarrons en boucle également. Il s'agit ici d'un exemple très simple de petit programme de test de charge. Le but est de vous montrer qu'il s'agit d'un développement relativement aisé, ce qui devrait vous encourager à réaliser ce genre de test très important pour évaluer les capacités de votre système.

Supervision avec les outils Linux

Linux comporte un certain nombre d'outils de supervision qui vous permettront de surveiller les performances de vos bases NoSQL. Le plus utile est `sysstat`, une collection d'outils de supervision de performances développés par un Français nommé Sébastien Godard (http://sebastien.godard.pagesperso-orange.fr/). `sysstat` peut être installé sur Debian et Ubuntu par un paquet des dépôts officiels :

```
sudo apt-get install sysstat
```

Lorsque le paquet est installé, vous avez à disposition plusieurs outils qui vous fournissent des indications sur l'état du système.

La commande iostat

La commande `iostat` retourne des statistiques de charge disque, cumulées depuis le démarrage du système. L'exemple suivant correspond au résultat d'un appel de la commande `iostat` sans qu'aucun paramètre ne soit mentionné.

```
Linux 3.0.0-29-generic-pae (ubuntu-server-64-fr)        10/02/2013      _i686_  (1
CPU)

avg-cpu: %user   %nice %system %iowait  %steal   %idle
           1,62    0,12    1,02    0,40    0,00   96,86

Device:     tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
scd0       0,00         0,00         0,00         48          0
sda       18,31        76,36        82,08   11397422   12252281
dm-0       5,06        38,24        37,50    5708357    5596904
dm-1      20,63        38,07        44,44    5682936    6633612
```

Les colonnes retournées sont :

- `Device` : le disque physique tel qu'il apparaît dans `/dev` ;
- `tps` : nombre de transactions par seconde, lectures et écritures cumulées ;
- `kB_read/s` : nombre de Ko lus par seconde ;
- `kB_wrtn/s` : nombre de Ko écrits par seconde ;
- `kB_read` : nombre total de Ko lus ;
- `kB_wrtn` : nombre total de Ko écrits.

dm-0
Les disques qui commencent par `dm-` sont des disques logiques créés par LVM *(Logical Volume Manager)*.

Vous pouvez aussi suivre l'activité disque entre deux appels de `iostat` en mentionnant un nombre d'exécutions en paramètre. Voici un exemple d'appel avec quelques options intéressantes :

```
iostat -mtx sda -d 2 10
```

Les options sont les suivantes :

- `m` : affiche les statistiques en Mo et non en Ko ;
- `t` : affiche le temps de chaque exécution ;
- `x` : affiche des informations étendues (option disponible depuis la version 2.5 du noyau Linux) ;
- `sda` : ne récupère les statistiques que pour ce périphérique ;
- `-d 2 10` : récupère les statistiques toutes les deux secondes, dix fois au total.

Les premières lignes de résultat sont représentées sur la figure 15-1.

Figure 15-1
Résultats de la commande iostat

```
rudi@ubuntu-server-64-fr:~/xml$ iostat -mtx sda -d 2 10
Linux 3.0.0-29-generic-pae (ubuntu-server-64-fr)        10/02/2013      _i686_  (1 CPU)

10/02/2013 08:12:14
Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,89     6,73   10,73    7,71     0,07     0,08    17,31     0,14    7,43    0,33   17,32   0,19   0,36

10/02/2013 08:12:16
Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    4,00    0,00     0,02     0,00     8,00     0,00    0,50    0,50    0,00   0,50   0,20

10/02/2013 08:12:18
Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,50    0,50    1,51     0,00     0,01     8,00     0,00    2,00    4,00    1,33   2,00   0,40

10/02/2013 08:12:20
Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               1,00     0,00    3,00    0,00     0,02     0,00    10,67     0,00    0,00    0,00    0,00   0,00   0,00

10/02/2013 08:12:22
Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,50    0,00    1,51     0,00     0,01     0,00     0,00    0,00    0,00    0,00   0,00   0,00
```

Les colonnes supplémentaires sont :

- `avgrq-sz` : taille moyenne des requêtes en nombre de secteurs du disque ;
- `avgqu-sz` : longueur moyenne de file d'attente du disque ;

- `r_await` : temps moyen d'attente en millisecondes pour les lectures, incluant l'attente dans la file et l'exécution ;
- `w_await` : temps moyen d'attente en millisecondes pour les écritures, incluant l'attente dans la file et l'exécution ;
- `svctm` : temps moyen du service, cette valeur doit être ignorée car elle n'est plus correcte ;
- `%util` : pourcentage de temps CPU où des demandes ont été effectuées sur le disque.

Avec ces indications, vous possédez tout ce dont vous avez besoin pour mesurer l'activité de vos disques et la pression que fait peser votre moteur NoSQL à un moment donné sur votre sous-système disque. Surveillez surtout la file d'attente (`avgqu-sz`) et le pourcentage d'utilisation. Une file d'attente supérieure à 0 indique que des processus attendent que le disque se libère. Si cette valeur est constamment au-dessus de zéro, songez à libérer le disque, en ajoutant de la RAM si les données peuvent être cachées, ou en répartissant les données sur plusieurs disques. Un pourcentage d'utilisation approchant les 100 % indique aussi une saturation.

La commande iftop

Nous avons vu comment surveiller l'activité disque. Un serveur de bases de données transmet ses données aux clients à travers le réseau, il est donc également important de surveiller l'activité des interfaces réseau. Outre les compteurs SNMP traditionnels, vous pouvez suivre en local l'activité réseau grâce à quelques outils en ligne de commande. Nous allons quitter provisoirement les outils `sysstat` pour installer `iftop`, qui réalise pour les interfaces réseau la même chose que la commande `top` pour les processus : afficher en temps réel le trafic le plus important. Nous allons installer `iftop` sur notre système Ubuntu :

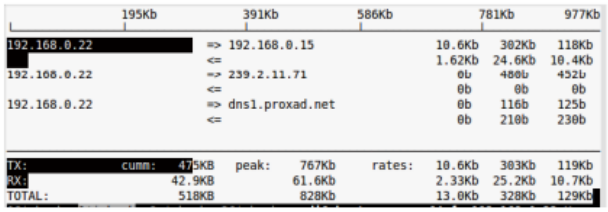
```
sudo apt-get install iftop
```

puis nous l'exécutons :

```
sudo iftop -i eth0
```

Afin de pouvoir accéder au trafic de l'interface `eth0`, il faut que `iftop` soit exécuté avec des privilèges administrateur, c'est pourquoi nous le lançons avec `sudo`. Nous avons limité l'interface avec l'option `-i`, ce qui nous permet de cibler seulement l'interface publique à travers laquelle passe le trafic des clients. La figure 15-2 donne une idée de l'affichage de la commande.

Figure 15-2
Résultats de la commande iftop



`TX` signifie transmission, donc envoi, et `RX` signifie réception.

Pour quitter l'affichage interactif, appuyez sur la touche `Q`.

La commande pidstat

La commande `pidstat` retourne des informations sur un processus. Vous pouvez l'utiliser pour surveiller différents aspects de votre moteur NoSQL, grâce aux options suivantes :

- `-d` : informations sur l'activité disque ;
- `-r` : utilisation mémoire et fautes de pages ;
- `-s` : utilisation de la pile *(stack)* ;
- `-u` : utilisation du CPU ;
- `-w` : changements de contextes.

De plus, l'option `-C` permet de filtrer les processus par expression rationnelle pour ne conserver que les processus souhaités. Voyons un premier exemple :

```
pidstat -C redis-server -d -p ALL
```

qui retourne l'activité disque pour `redis-server` depuis le démarrage du système. Nous indiquons `-p ALL` pour afficher tous les processus, même ceux pour lesquels aucune statistique n'est à afficher. Nous obtenons une ligne avec des statistiques à 0 si le processus n'a pas encore eu d'activité. Le résultat est le suivant :

```
Linux 3.0.0-29-generic-pae (ubuntu-server-64-fr)        10/02/2013      _i686_  (1 CPU)

10:35:23          PID   kB_rd/s   kB_wr/s kB_ccwr/s  Command
10:35:23        32155      1,96      0,10      0,00  redis-server
```

Ces informations correspondent à ce qu'on peut obtenir avec `iostat`.

L'option `-r` donne des informations sur l'occupation mémoire et les fautes de pages. Une faute de page indique un défaut de mémoire : le processus cherche à accéder à une page de mémoire dans l'espace d'adressage virtuel (la RAM rendue disponible par le système) à l'aide de l'adresse fournie par le système d'exploitation, et cette page n'est pas présente à l'adresse désignée. Il faut donc que le système la récupère. Si la faute est majeure *(major page fault)*, la page n'est pas présente en mémoire, mais sur le disque (en swap), il faut donc la remonter en mémoire. Un excès de fautes de pages majeures indique probablement qu'il n'y a pas assez de mémoire dans le système. Cette indication est par exemple intéressante pour MongoDB, qui s'appuie pour son système de cache sur les fonctionnalités du système d'exploitation. Voici un exemple d'appel :

```
pidstat -C mongod -r -p ALL
```

et son résultat :

```
Linux 3.0.0-29-generic-pae (ubuntu-server-64-fr)        10/02/2013      _i686_  (1 CPU)
```

```
10:44:52    PID  minflt/s  majflt/s     VSZ    RSS  %MEM  Command
10:44:52  13524      0,13      0,01  148928   1520   0,15  mongod
```

Les colonnes signifient :

- `minflt/s` : nombre de fautes de pages mineures par seconde ;
- `majflt/s` : nombre de fautes de pages majeures par seconde ;
- `VSZ` : taille en mémoire virtuelle (la RAM allouée par le système au processus) en Ko ;
- `RSS` *(Resident Set Size)* : taille de mémoire physique, non swappée, utilisée par le processus et exprimée en Ko ;
- `%MEM` : pourcentage de mémoire physique utilisée par le processus.

La différence entre `VSZ` et `RSS` indique la partie de la mémoire du processus qui est dans le swap. Cette occupation mémoire inclut l'utilisation de bibliothèques partagées, qui peuvent aussi être utilisées par d'autres processus. Vous pouvez obtenir des détails d'occupation mémoire à l'aide de la commande `pmap` (voir chapitre 14, page 274).

La commande sar

`sysstat` installe un démon qui collecte des statistiques d'utilisation à travers le temps. Pour activer cette collecte, éditez le fichier de configuration `/etc/default/sysstat` et changez

```
ENABLED="false"
```

en

```
ENABLED="true"
```

Les statistiques seront collectées dans `/var/log/sysstat/` et pourront être interrogées avec la commande `sar`. Les options de la commande `sar` sont :

- `-B` : statistiques de pagination ;
- `-b` : statistiques d'entrées-sorties ;
- `-d` : statistiques disque pour chaque bloc présent dans `/dev` ;
- `-H` : statistiques d'utilisation des hugepages (voir encadré suivant) ;
- `-m` : statistiques de gestion d'énergie ;
- `-n` : statistiques réseau ;
- `-q` : statistiques sur les files d'attente et de la charge du système ;
- `-R` : statistiques mémoire ;
- `-r` : statistiques d'utilisation de la mémoire ;
- `-u` : utilisation des CPU ;
- `-W` : statistiques de swap ;
- `-w` : créations de tâches et switchs du système.

Les hugepages

Les hugepages sont des zones de mémoire qui sont allouées via la fonction système `mmap()` a des tailles bien plus importantes que les pages de mémoire traditionnelles. Cette allocation permet de meilleures performances car la mémoire est accédée plus rapidement, sans qu'il soit nécessaire de recourir à des tables de traduction de l'adresse mémoire. Par ailleurs, ces pages sont fixées en mémoire et ne peuvent être swappées. Les moteurs basés sur Java peuvent bénéficier de ces avantages en configurant l'option `UseLargePageSize` de la JVM.

Voici un exemple d'appel de la commande `sar` :

```
sar -q 1 3
```

qui demande des statistiques de file d'attente pour un intervalle d'une minute, sur les trois dernières valeurs enregistrées. Voici le résultat :

```
Linux 3.0.0-29-generic-pae (ubuntu-server-64-fr)        10/02/2013      _i686_  (1 CPU)

17:37:55  runq-sz  plist-sz   ldavg-1   ldavg-5  ldavg-15   blocked
17:37:56         0       264      0,00      0,02      0,09         0
17:37:57         0       264      0,00      0,02      0,09         0
17:37:58         0       264      0,00      0,02      0,09         0
Moyenne :        0       264      0,00      0,02      0,09         0
```

Nous voyons qu'il y avait 264 tâches actives (`plist-sz`). La charge moyenne du système pour les cinq dernières minutes (`ldavg-5`) était de 0,02, et de 0,09 durant les quinze dernières minutes. Cela représente 2 % et 9 % de charge, 1 correspondant à une charge de 100 % du processeur.

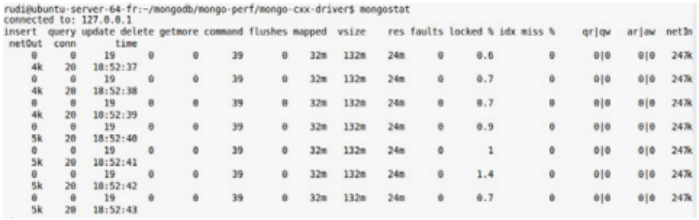
Supervision avec les outils intégrés

Les moteurs comportent souvent leurs propres outils de monitoring, qui vous permettent de suivre l'activité en récupérant des compteurs internes spécifiques. Voici quelques exemples sur nos moteurs.

MongoDB

MongoDB inclut la commande `mongostat`, qui retourne en permanence le nombre de requêtes exécutées et d'autres indicateurs. Il suffit d'exécuter la commande `mongostat` dans un shell. La figure 15-3 donne un exemple de quelques lignes de résultats.

Figure 15-3
Résultats obtenus avec la commande mongostat



```
rudi@ubuntu-server-64-fr:~/mongodb/mongo-perf/mongo-cxx-driver$ mongostat
connected to: 127.0.0.1
insert  query update delete getmore command flushes mapped  vsize    res faults locked % idx miss %     qr|qw   ar|aw  netIn
 netOut  conn       time
    0     0     19      0       0      39       0   32m   132m    24m      0      0.6          0       0|0     0|0   247k
    4k    20  18:52:37
    0     0     19      0       0      39       0   32m   132m    24m      0      0.7          0       0|0     0|0   247k
    4k    20  18:52:38
    0     0     19      0       0      39       0   32m   132m    24m      0      0.7          0       0|0     0|0   247k
    4k    20  18:52:39
    0     0     19      0       0      39       0   32m   132m    24m      0      0.9          0       0|0     0|0   247k
    5k    20  18:52:40
    0     0     19      0       0      39       0   32m   132m    24m      0        1          0       0|0     0|0   247k
    5k    20  18:52:41
    0     0     19      0       0      39       0   32m   132m    24m      0      1.4          0       0|0     0|0   247k
    5k    20  18:52:42
    0     0     19      0       0      39       0   32m   132m    24m      0      0.7          0       0|0     0|0   247k
    5k    20  18:52:43
```

Les indicateurs sont les suivants :

- `insert` : nombre d'objets insérés par seconde ;
- `query` : nombre de requêtes par seconde ;
- `update` : nombre d'objets modifiés par seconde ;
- `delete` : nombre d'objets supprimés par seconde ;
- `getmore` : nombre d'appels de curseurs sur des objets ouverts, par seconde ;
- `command` : nombre de commandes par seconde ;
- `flushes` : nombre de `fsync()` par seconde, donc de flush de la mémoire vers le disque ;
- `mapped` : total des données mappées en mémoire, en Mo ;
- `vsize` : taille de la mémoire utilisée par MongoDB ;
- `res` : taille de la mémoire résidente, non swappée ;
- `faults` : nombre de fautes de pages par seconde (voir la section précédente sur la commande `pidstat`, page 296) ;
- `locked%` : pourcentage de temps utilisé où un verrou global d'écriture est posé ;
- `idx miss%` : pourcentage de tentatives d'accès à une page d'index qui a déclenché une faute, donc où la page d'index n'était pas en mémoire et a dû être chargée depuis le disque ;
- `qr|qw` : longueur des files d'attente de lecture (r) et d'écriture (w) ;
- `ar|aw` : nombre de clients en train de lire (r) et d'écrire (w) ;
- `NetIn` : trafic réseau entrant en octets ;
- `NetOut` : trafic réseau sortant en octets ;
- `conn` : nombre de connexions ouvertes.

Cassandra

Cassandra inclut l'outil `nodetool`, qui permet d'administrer certains de ses éléments et de récupérer des informations. Les commandes suivantes sont utiles pour la supervision :

- `nodetool ring` : retourne des informations sur l'anneau des nœuds ;
- `nodetool info` : retourne des informations sur le nœud sur lequel il est exécuté ;

- `nodetool cfstats` : retourne des statistiques sur les familles de colonnes ;
- `nodetool tpstats` : retourne des informations sur les threads ;
- `nodetool netstats` : retourne des informations sur l'activité réseau ;
- `nodetool compactionstats` : retourne des informations sur les opérations de compactage.

Voici un exemple d'appel de `nodetool info` :

```
Token            : 50296331533206892393222798876815311727
Gossip active    : true
Thrift active    : true
Load             : 316.99 MB
Generation No    : 1360329584
Uptime (seconds): 190262
Heap Memory (MB): 346,26 / 478,00
Data Center      : datacenter1
Rack             : rack1
Exceptions       : 0
Key Cache        : size 480 (bytes), capacity 24117216 (bytes), 13 hits, 23 requests,
0,565 recent hit rate, 14400 save period in seconds
Row Cache        : size 0 (bytes), capacity 0 (bytes), 0 hits, 0 requests, NaN recent
hit rate, 0 save period in seconds
```

Nous obtenons l'état du nœud, la taille occupée en mémoire, la taille et les statistiques du cache des clés et du cache des lignes. En combinant les différentes commandes `nodetool`, vous pourrez effectivement surveiller l'activité de vos nœuds Cassandra.

Redis

Redis comporte la commande `monitor` qui permet de suivre son activité en temps réel, sous forme de trace de requêtes. Nous pouvons la lancer avec `redis-cli`, de la manière suivante :

```
redis-cli monitor
```

Voici un extrait du résultat :

```
1360534695.969207 [0 127.0.0.1:48804] "SET" "300730417307320320" "@Adrienbrt @flobrager
@florianelcrt je redis bref !"
1360534697.503696 [0 127.0.0.1:48895] "INFO"
1360534717.449005 [0 127.0.0.1:48902] "INFO"
1360534735.673057 [0 127.0.0.1:48804] "SET" "300730583879938049" "Ej jutro musze isc
na 5 godzin do sql, jechac do ortodonty, do kosmetyczki i na korki z chemii. \xc5\
xbbal dupe sciska nie?"
1360534737.458720 [0 127.0.0.1:48914] "INFO"
1360534743.703835 [0 127.0.0.1:48804] "SET" "300730617501458432" "Startups that have
chosen sql... how do you keep your site online nonstop when performing updates?: I
dont know ... http://t.co/EvxUIwQ2"
1360534752.678744 [0 127.0.0.1:48804] "SET" "300730655095001088" "Personne me respecte
ici j'vous l'dis , elle m'le redit."
1360534757.535845 [0 127.0.0.1:48922] "INFO"
```

```
1360534777.461079 [0 127.0.0.1:48929] "INFO"
1360534792.001277 [0 127.0.0.1:48804] "SET" "300730820111499264" "SQL 2008 Forum - SQL
Server Command line utility for windows XP: Hello, I am trying to run a job lying on a
SQL S... http://t.co/g2DX1q4W"
1360534795.141437 [0 127.0.0.1:48804] "SET" "300730833348751361" "RT @Mehdi_Carter:
Personne me respecte ici j'vous l'dis , elle m'le redit."
```

Comme vous pouvez le constater, chaque commande reçue par Redis est montrée telle quelle. La première partie du résultat correspond au timestamp de la machine.

Performances de monitor

L'exécution de la commande `monitor` peut diminuer les performances de Redis de 50 %. En guise d'exemple, consultez le test reproduit sur la page d'aide de la commande monitor à l'adresse suivante: http://redis.io/commands/monitor.

Nous avons également vu au chapitre 10 que Redis inclut un *slowlog*, qui conserve une trace des requêtes estampillées lentes.

Supervision avec Ganglia

Ganglia (http://ganglia.sourceforge.net/) est un système de supervision qui se distingue de ses concurrents libres comme Nagios par ses capacités de montée en charge. Conçu dès le départ pour assurer la supervision d'architectures fortement distribuées (en *grid*), il permet de surveiller des dizaines de milliers de machines. Développé à l'université de Berkeley, il est utilisé par un grand nombre d'entreprises de taille importante qui doivent gérer un très grand parc de machines, comme Twitter, Flickr, Reuters, Boeing et même Microsoft.

Comme son nom l'indique (« ganglia » est le pluriel de ganglion, le tissu interconnecté du système nerveux), Ganglia est distribué et décentralisé. Son architecture ressemble conceptuellement aux systèmes décentralisés comme Dynamo ou Cassandra. Afin de pouvoir monter en charge sans problème, chaque machine sur laquelle Ganglia est installé interroge elle-même les données locales. Elle les envoie ensuite à des collecteurs de statistiques, potentiellement redondants, dans un cluster de machines à travers une adresse multicast par défaut. À chaque niveau du fonctionnement de Ganglia, les performances ont été soignées pour que cette activité ne charge pas de façon significative les nœuds et le réseau. La communication se fait via UDP à travers une adresse multicast. Ganglia comporte nativement toutes les métriques nécessaires pour surveiller le système et le réseau, ainsi qu'un environnement Hadoop, HDFS et Hbase à l'aide de sa capacité à récupérer des métriques JMX.

JMX

JMX *(Java Management Extensions)* est une API pour Java permettant de gérer le fonctionnement d'une application Java en cours d'exécution. JMX est intégré dans J2SE depuis la version 5.0. Attention, JMX est désactivé par défaut dans la version 5.0. Utilisez `java -Dcom.sun.management.jmxremote`.

Des modules de statistiques peuvent être développés en Python ou C et être ajoutés à Ganglia.

Ganglia est composé de plusieurs modules qui assurent différentes tâches permettant la distribution du travail de récupération des compteurs. Le démon `gmond` recueille les mesures. Le démon `gmetad` recueille les statistiques. Une application web (démon `gweb`) permet d'afficher ces statistiques et d'accumuler des graphes à l'aide de `RRDTool`, une bibliothèque de stockage et d'affichage de données journalisées dans le temps.

Installer Ganglia

Sur Ubuntu, la façon la plus simple d'installer Ganglia consiste à utiliser le paquet de la distribution officielle :

```
sudo apt-get install ganglia-monitor
```

qui installe `gmond`, le démon de supervision, dont vous aurez besoin sur la plupart des nœuds.

Bogue du paquet ganglia-monitor

Ce paquet peut boguer sur votre distribution et afficher une erreur pendant la phase de configuration de dpkg (le gestionnaire de paquets): `useradd: group ganglia exists - if you want to add this user to that group, use -g`. Si vous rencontrez cette erreur, lancez la commande suivante pour ajouter le compte `ganglia` dans le groupe: `sudo useradd ganglia -g ganglia`. Relancez ensuite l'installation du paquet.

Le démon est automatiquement démarré par l'installation. Le fichier de configuration se trouve pour nous dans `/etc/ganglia/gmond.conf`. Nous allons changer le nom du cluster, qui est « unspecified » par défaut.

Nous installons ensuite l'interface web sur notre machine. Dans une installation en cluster, vous installerez bien sûr ce paquet sur une seule machine, celle sur laquelle vous vous connecterez pour visualiser vos rapports.

```
sudo apt-get install ganglia-webfrontend
```

Ensuite, nous devons ajouter le fichier de paramétrage à la configuration d'Apache. Ce fichier, situé dans le répertoire `/etc/ganglia-webfrontend/apache.conf`, contient une seule ligne, qui crée l'alias vers le répertoire où l'application web est installée. Nous créons un lien symbolique vers ce fichier dans la configuration d'Apache et nous redémarrons Apache pour que cette modification soit prise en compte :

```
sudo ln -s /etc/ganglia-webfrontend/apache.conf /etc/apache2/sites-enabled/ganglia.conf
sudo apache2ctl restart
```

Nous vérifions ensuite l'adresse http://serveur/ganglia en la saisissant dans la barre d'adresses de notre navigateur web.

Voyons maintenant comment ajouter des compteurs spécifiques à nos moteurs NoSQL.

Ajouter des modules Python

Depuis sa version 3.1, Ganglia permet de créer des modules de récupération de compteurs externes et des modules Python. Un certain nombre de modules pour des outils comme MongoDB, Redis, CouchDB et ElasticSearch sont disponibles sur GitHub à l'adresse suivante : https://github.com/ganglia/gmond_python_modules/t.

Nous devons modifier le fichier de configuration `/etc/ganglia/gmond.conf` pour ajouter la bibliothèque qui gère les modules Python. Nous ajoutons la section suivante dans la section `modules` :

```
modules {
  ...
  module {
    name = "python_module"
    path = "/usr/lib/ganglia/modpython.so"
    params = "/usr/lib/ganglia/python_modules/"
  }
}
```

`gmond.conf` contient aussi la ligne suivante :

```
include ('/etc/ganglia/conf.d/*.conf')
```

C'est donc à cet endroit que nous allons déposer les fichiers `.conf` de ces modules. Certains fichiers de configuration Python ont l'extension `.pyconf`, nous ajoutons donc cette ligne dans `gmond.conf` :

```
include ('/etc/ganglia/conf.d/*.pyconf')
```

Nous récupérons ensuite les modules grâce à `git` :

```
git clone git://github.com/ganglia/gmond_python_modules.git
```

puis nous créons les répertoires et nous copions les fichiers :

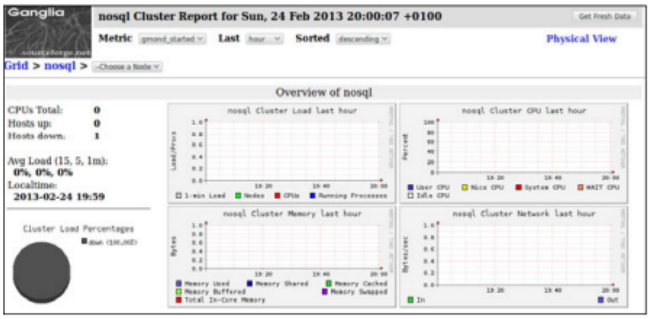
```
sudo mkdir /etc/ganglia/conf.d/
sudo mkdir /usr/lib/ganglia/python_modules/
cd gmond_python_modules/
sudo cp mongodb/conf.d/mongodb.conf /etc/ganglia/conf.d/
sudo cp mongodb/python_modules/mongodb.py /usr/lib/ganglia/python_modules/
sudo cp redis-gmond/conf.d/redis-gmond.pyconf /etc/ganglia/conf.d/
sudo cp redis-gmond/python_modules/redis-gmond.py /usr/lib/ganglia/python_modules/
sudo cp couchdb/conf.d/couchdb.pyconf /etc/ganglia/conf.d/
sudo cp couchdb/python_modules/couchdb.py /usr/lib/ganglia/python_modules/
```

Nous redémarrons ensuite `gmond` :

```
sudo service ganglia-monitor restart
```

Après quelque temps, nos compteurs apparaissent sur l'interface web de Ganglia, à l'adresse http://serveur/ganglia. Un exemple est reproduit sur la figure 15-4.

Figure 15-4
Interface web de Ganglia



16

Étude de cas : le NoSQL chez Skyrock

Afin d'illustrer l'utilisation des moteurs NoSQL, nous avons choisi de prendre pour exemple la chaîne de radio Skyrock, bien présente sur le Web, notamment à travers les Skyblogs. Pour ses différents besoins Internet, Skyrock a recours à des moteurs NoSQL depuis quelques années.

Avant même d'avoir opté pour des solutions « officiellement » NoSQL, les développeurs de Skyrock, dont la politique a toujours été de faire appel à des outils libres, utilisaient le classique tandem PHP-MySQL dans une architecture LAMP (Linux, Apache, MySQL, PHP). Cette architecture constitue toujours la base du stockage des données du site skyrock.com, qui compte à ce jour environ 200 serveurs MySQL et memcached. Mais même lorsque les développeurs de Skyrock ne faisaient appel qu'à MySQL, il s'agissait d'une utilisation plutôt non relationnelle du moteur, en évitant les jointures et en stockant l'équivalent de paires clé-valeur dans les tables MySQL. Tous les accès aux données contenues dans chaque table s'effectuent par l'intermédiaire de la clé, définie comme clé primaire de la table, sans utilisation de recherche sur d'autres colonnes par indexation secondaire. Pour assurer la montée en charge, les développeurs de Skyrock ont commencé à « casser le relationnel » encore plus, en séparant tout d'abord les bases de données par service (une base pour les blogs, une base pour les articles, une base pour les commentaires, etc.), ce qui leur a permis d'utiliser des serveurs différents pour chaque élément. Cela veut dire bien sûr que les jointures ont disparu de leur code SQL, et qu'elles ont dû être réalisées du côté du client. Pour assurer ensuite de nouvelles montées en charge horizontales, ils ont partitionné (shardé) à partir de 2007 les données de chaque service sur dix machines, sur des plages de clés, en utilisant la technique classique du modulo. À l'époque, l'idée du hachage consistant n'était pas très répandue, et comme ils ne comptaient pas ajouter très régulièrement de nouvelles machines, cela n'a pas posé de problème en pratique.

Mais, vous direz-vous, comment effectuer des recherches ? Par exemple, si les articles de blog sont retrouvés par la clé (comme un identifiant d'article), comment faire pour rechercher les articles par mot-clé, par sujet ou par date de publication ? Tout d'abord, notons que certaines recherches ne sont pas pertinentes sur le site skyrock.com, telles que le regroupement d'articles de blog par mois, comme cela est souvent proposé sur les plates-formes de publication de blogs. Pour le reste, les performances sont assurées par le cache en mémoire et la création de listes inversées. Par exemple, pour distinguer les articles publiés des brouillons, la liste des articles d'un utilisateur est récupérée par sa clé et stockée dans le cache de memcached. Le filtre – équivalent de la clause `WHERE` dans une requête SQL – est effectué par une boucle de recherche dans le code client, et reste rapide grâce au stockage en cache. Cette approche implique une dénormalisation et donc une duplication des données, ce qui augmente la taille du cache, mais cette solution est tout à fait satisfaisante pour Skyrock.

Le développement de solutions en interne

La question du maintien de la session utilisateur s'était aussi posée. Habituellement, une session sur un site web fait appel à la fois au client et au serveur : du côté de l'utilisateur, un cookie maintient un UUID de session généré par PHP, et du côté du serveur, cet UUID est maintenu, par exemple en base de données. Pour permettre au traitement serveur d'être plus rapide, Frank Denis (à qui l'on doit notamment le serveur FTP libre PureFTPD) a développé en 2004 un outil de maintien de sessions du côté serveur pour Skyrock nommé Sharedance. Le projet est libre et peut être téléchargé à l'adresse suivante : http://www.pureftpd.org/project/sharedance.

Sharedance est constitué d'un démon écrit en C qui écoute sur un port TCP les requêtes de clients qui lui envoient des demandes d'écriture et de lecture de paires clé-valeur, et qui les maintient sur un système de fichiers, équivalent simplifié de memcached (Sharedance a été publié pour la première fois en 2004, memcached en 2003). Afin d'assurer la partie cache, il suffit de faire écrire Sharedance sur un *ramdisk* tel que tmpfs sur Linux.

> **Le ramdisk**
>
> Le ramdisk est une partition visible par le système de fichier et qui est en réalité une zone de la mémoire vive. C'est donc la simulation d'un disque en RAM, pour offrir de grandes performances d'accès à des fichiers.

Le maintien en RAM est ainsi assuré à moindre frais. Les opérations d'écriture et de lecture des paires sont très simples. Le code source de Sharedance implémente le minimum nécessaire pour permettre à du code PHP de travailler avec lui :

```
sharedance_store($key, $data)
sharedance_fetch($key)
sharedance_delete($key)
```

en d'autres termes, de simples `GET` et `SET`.

Topy

En 2008, un autre besoin s'est fait sentir : établir et afficher des classements d'utilisateurs en fonction de différents critères : nombre de visites, âge, sexe, etc. Ces listes ont d'abord été générées en base de données, dans des tables de résultats, par des requêtes planifiées. Mais pour assurer des calculs statistiques plus rapides, Nicolas Vion, l'un des développeurs de Skyrock, a créé un outil nommé Topy. Le code source en C++ de Topy et son extension PHP sont disponibles à l'adresse suivante : https://github.com/zmoo/topy. Topy est un serveur qui répond en TCP ou en UDP, et dont la tâche consiste à générer des statistiques et des classements en temps réel. Pour ce faire, il manipule des paires clé-valeur, la clé étant, par exemple, une clé d'utilisateur ou tout type de valeur sur laquelle des classements doivent être effectués, et la valeur est une structure qui contient des agrégats sur lesquels des opérations simples peuvent être réalisées, comme un `add`. Sur skyrock.com, Topy est utilisé pour tous types de classements (visites, votes, tags, « kif », thèmes, etc.) et également pour des statistiques internes. La figure 16-1 montre un fragment de page de profil utilisateur où Topy est mis en œuvre.

Figure 16-1
Page de profil Skyrock qui utilise Topy

L'outil a évolué avec le temps pour intégrer de nouveaux types de clés, par exemple pour les classements de vidéos. L'extension Topy pour PHP implémente aussi une gestion de pool de connexions persistantes et une désérialisation du flux entre le serveur et le client PHP, en format *PHP serialize*. Le but est d'optimiser les performances à tous les niveaux de la chaîne.

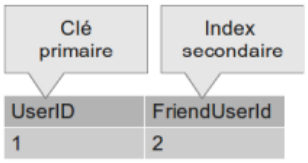
Fluxy

En 2009, afin de gérer les listes d'articles des skyblogs, Nicolas Vion a développé un autre outil, Fluxy, disponible à cette adresse : http://fluxy.tuxfamily.org/. Comme il est indiqué sur la page de Fluxy, l'outil est toujours en production chez Skyrock et permet de gérer des listes d'articles de plusieurs dizaines de millions d'utilisateurs sur deux machines, une active et une passive pour la redondance. Fluxy maintient en mémoire des listes d'articles par clé d'utilisateur. Les listes sont finies et utilisent un algorithme FIFO (*First In, First Out* : premier entré, premier sorti) pour conserver un nombre maximal d'articles. Les articles sont conservés par ordre chronologique, avec un certain nombre de champs dans la structure : date, type, tags et corps de l'article notamment. Fluxy gère donc une liste d'articles. Le besoin de Skyrock étant d'afficher une liste d'articles publiés par les contacts ou amis d'un utilisateur, il faut agréger ensuite ces listes pour afficher

les dernières publications d'amis. C'est Fluxy qui s'occupe de cette opération, en récupérant une liste d'amis par la clé de l'utilisateur, puis en en bâtissant une liste pondérée d'articles rédigés par ces amis. La liste est simple et ne contient pas le texte ou les ressources de l'article. Chaque appel sur un élément de cette liste (par exemple, l'utilisateur voulant accéder à un de ces articles) fera l'objet d'un appel à la ressource contenue dans MySQL et cachée dans memcached.

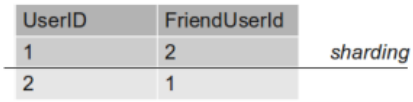
La liste d'amis elle-même est stockée dans MySQL mais de façon non relationnelle. Si vous maintenez une relation entre des utilisateurs dans une base de données relationnelle, votre réflexe est de créer une table de correspondance entre deux identifiants d'utilisateurs. C'est bien entendu ce qu'a fait Skyrock. Si l'amitié est bidirectionnelle, c'est-à-dire qu'en étant ami avec quelqu'un, cette autre personne est automatiquement amie avec vous, vous penserez à créer simplement un index secondaire sur la colonne identifiant l'ami, comme illustré sur la figure 16-2.

Figure 16-2
La table d'amitiés



Il s'agit de l'approche relationnelle traditionnelle. Mais ce mécanisme présente un inconvénient : si vous comptez sharder votre table sur la clé de l'utilisateur, les recherches par l'index secondaire seront impossibles ou devront être effectuées sur tous les shards, ce qui est impraticable. C'est pour cette raison que les développeurs de Skyrock ont décidé de maintenir une liste dupliquée d'amis, comme illustrée sur la figure 16-3.

Figure 16-3
La capacité de sharder la table d'amitiés



Toutes les recherches se font donc sur la clé primaire, on retrouve effectivement une pure approche clé-valeur.

L'utilisation de Redis

MySQL reste la solution de stockage à long terme la plus utilisée chez Skyrock. En ce qui concerne l'accélération des accès et la manipulation de données plus volatiles, Redis a été choisi pour plusieurs développements récents. Redis est vraiment intéressant pour s'assurer des latences les plus faibles en accès et manipulation de données. En 2010, les développeurs de Skyrock ont travaillé sur la manière la plus efficace possible d'afficher les utilisateurs en ligne sur skyrock.com (figure 16-4).

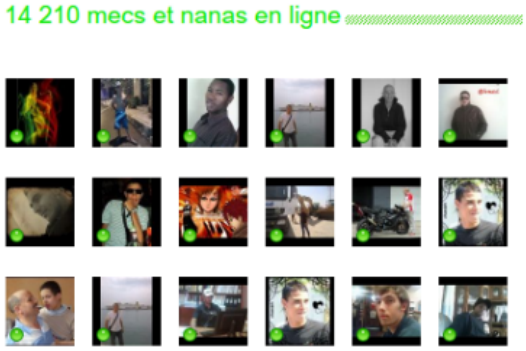


Figure 16-4
Un fragment du site Skyrock utilisant Redis

Comment gérer une liste aussi dynamique ? Les développeurs sont d'abord partis sur une approche MySQL et memcached, qui a fonctionné pendant quelques temps. Afin d'obtenir de meilleures performances et de s'assurer de pouvoir tenir la charge dans le futur, ils ont développé en parallèle une solution utilisant Redis. Le principe est simple : la liste des utilisateurs en ligne est rafraîchie en temps réel et partitionnée toutes les minutes. Cela permet donc de maintenir dans Redis une clé qui correspond à une minute. Dans la valeur, un ensemble (set) Redis est maintenu avec la liste des identifiants d'utilisateurs en ligne.

Si vous vous connectez sur skyrock.com et que vous souhaitez afficher la liste de vos amis en ligne, le code crée une clé temporaire sur votre identifiant, incluant un set de votre liste d'amis. Il suffit ensuite de faire une intersection des deux sets pour retrouver vos amis en ligne. Voici en quelques lignes de code, que nous saisissons dans l'invite interactive de Redis (`redis-cli`), le principe de la recherche :

```
SELECT 1
SADD 20121012T12:53 10 24 76 934 512 534 954 123 404 65
SMEMBERS 20121012T12:53
SADD 734 523 76 834 634 123 811 450 512
SINTER 734 20121012T12:53
PEXPIRE 734 1
```

La première ligne sélectionne une base de données autre que celle par défaut (`0`) pour isoler le travail. Ensuite, nous utilisons la commande `SADD` pour ajouter un set avec la clé `20121012T12:53`, c'est-à-dire un *datetime* à la granularité de la minute, en l'occurrence 12:53 le 12 octobre 2012. Nous ajoutons dix identifiants, nous aurions pu en ajouter 14 000. Nous allons essayer cela dans un instant, mais pour le moment, faisons notre essai avec ceci. Admettons que l'utilisateur 734 se connecte et que nous voulons afficher sa liste d'amis. Nous créons un set temporaire sur la clé 734 contenant les identifiants des amis, ici au nombre de huit. Il nous suffit ensuite de faire un `SINTER` sur les deux clés pour retourner l'intersection des deux sets, donc la liste d'amis en ligne. Finalement, nous affectons une période d'expiration de notre clé temporaire d'une milliseconde. Redis nettoiera la clé 734 pour nous. Les résultats de l'intersection sont illustrés sur la figure 16-5.

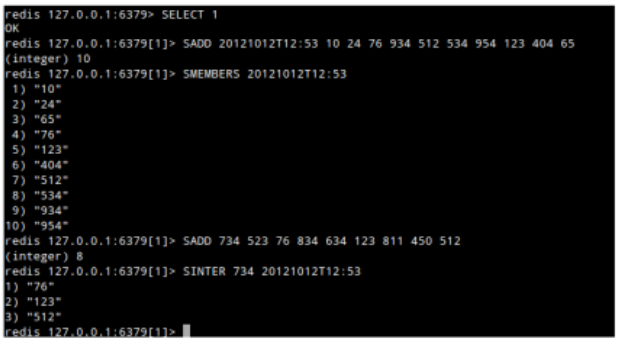


Figure 16-5
Résultat du calcul d'intersection dans Redis

Tester la charge

Essayons de voir comment se comporte cette solution avec une charge plus réaliste. Nous allons produire une clé comportant 14 000 identifiants d'utilisateurs, que nous allons également alourdir en créant des UUID. Avant cela, nous allons vider notre base de données `1` pour repartir à zéro :

```
SELECT 1
FLUSHDB
```

Ensuite, nous écrivons le code Python suivant :

```
import uuid
import redis
from datetime import datetime, time

r = redis.StrictRedis(host='localhost', port=6379, db=1)

cle = datetime.now().strftime("%Y-%m-%dT%H:%M")
user = uuid.uuid4()

pipe = r.pipeline(transaction=False)
pipe.sadd(cle, user)

for i in range(14000):
        id = uuid.uuid4()
        if i% 100 == 0:
                pipe.sadd(user, id)
        pipe.sadd(cle, id)

pipe.execute()
```

Nous utilisons ici le module `redis-py` pour accéder à Redis. Nous ouvrons une connexion sur la machine locale en indiquant la base de données `1`. Nous définissons la clé correspondant à la minute actuelle et nous créons un premier utilisateur grâce au module `uuid` de Python. Il

nous servira d'utilisateur témoin. Afin d'éviter les allers-retours vers Redis, nous utilisons un objet `pipeline` qui va permettre de transmettre les requêtes vers Redis par lots. Nous insérons 14 000 UUID dans la clé correspondant à la minute courante, et en utilisant un modulo 100, nous créons 140 amis dans la clé temporaire de notre utilisateur témoin. Afin de mesurer les performances de l'opération, nous utilisons la classe de profilage `cProfile` de Python, en appelant notre script (que nous avons nommé `skyrock.py`) ainsi :

```
python -m cProfile -o skyrock.profile skyrock.py
```

Environnement Python

Pour une fois, nous n'avons pas exécuté ce script dans notre environnement Python (`~/python_env/nosql/`) car un bogue actuel rend indisponible le module `urandom` dans l'environnement, nécessaire pour utiliser le module `uuid`.

Les informations de performances seront stockées dans le fichier `skyrock.profile` que nous analyserons ensuite. Voyons d'abord à l'aide du client Redis si tout s'est bien passé :

```
SELECT 1
KEYS *
```

Nous obtenons le résultat suivant :

```
1) «2012-10-12T15:28»
2) «59ddefc6-9799-4ae1-94c5-11d8711eea7b»
```

Nous avons bien nos deux clés. Vérifions le nombre d'éléments à l'aide de l'instruction `SCARD` qui retourne le nombre d'éléments d'un set :

```
SCARD "2012-10-12T15:28"
SCARD "59ddefc6-9799-4ae1-94c5-11d8711eea7b"
```

Nous obtenons bien 14 001 et 140 membres, respectivement. Il nous reste à réaliser l'intersection :

```
SINTER «59ddefc6-9799-4ae1-94c5-11d8711eea7b» «2012-10-12T15:28»
```

Le résultat est instantané et retourne les 140 UUID, comme nous le voyons sur la figure 16-6.

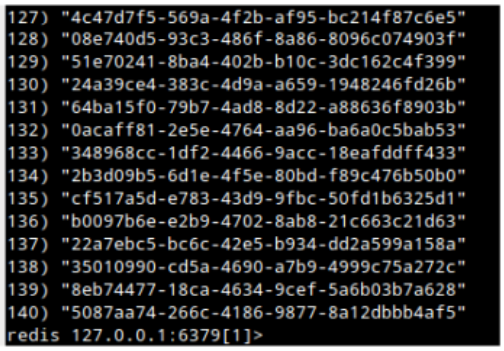


Figure 16-6
Résultat de l'intersection sur un ensemble de 14 000 membres

Voyons enfin l'occupation en mémoire à l'aide la commande `INFO` qui retourne pour notre exemple :

```
# Memory
used_memory:1639192
used_memory_human:1.56M
```

ce qui est très raisonnable. Voyons enfin les performances indiquées par Python. Nous ouvrons le fichier de profil à l'aide du module `pstats` de Python :

```
python -m pstats skyrock.profile
```

ce qui nous ouvre une invite d'analyse du fichier de profil. Nous affichons les temps d'exécution les plus hauts :

```
strip
sort time
stats 30
```

Ces commandes de `pstats` provoquent un affichage des trente opérations les plus coûteuses en termes de temps d'exécution du script. Le résultat est illustré sur la figure 16-7.

Figure 16-7
Analyse de temps d'exécution avec pstats

```
List reduced from 291 to 30 due to restriction <30>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 14005    0.241    0.000    0.305    0.000 uuid.py:101(__init__)
 14001    0.167    0.000    0.530    0.000 uuid.py:531(uuid4)
 14142    0.134    0.000    0.285    0.000 connection.py:324(pack_command)
 14142    0.074    0.000    0.149    0.000 socket.py:406(readline)
155890    0.069    0.000    0.069    0.000 {isinstance}
 42425    0.058    0.000    0.123    0.000 connection.py:312(encode)
 14142    0.056    0.000    0.247    0.000 connection.py:101(read_response)
 14004    0.053    0.000    0.053    0.000 {map}
 14001    0.043    0.000    0.059    0.000 __init__.py:52(create_string_buffer)
     1    0.034    0.034    1.341    1.341 skyrock.py:1(<module>)
 14281    0.031    0.000    0.031    0.000 uuid.py:197(__str__)
     5    0.029    0.006    0.029    0.006 {method 'read' of 'file' objects}
 14141    0.027    0.000    0.319    0.000 client.py:1762(parse_response)
 14141    0.024    0.000    0.038    0.000 client.py:1652(execute_command)
 14141    0.023    0.000    0.292    0.000 client.py:368(parse_response)
 14142    0.023    0.000    0.171    0.000 connection.py:61(read)
71144/71124    0.018    0.000    0.018    0.000 {len}
 14142    0.018    0.000    0.269    0.000 connection.py:301(read_response)
 14134    0.017    0.000    0.017    0.000 {method 'read' of 'cStringIO.StringO' objects}
     1    0.016    0.016    0.650    0.650 client.py:1745(_execute_pipeline)
 14150    0.016    0.000    0.016    0.000 {method 'write' of 'cStringIO.StringO' objects}
 56572    0.015    0.000    0.015    0.000 _compat.py:20(<lambda>)
 14141    0.014    0.000    0.053    0.000 client.py:934(sadd)
 14136    0.014    0.000    0.014    0.000 {method 'endswith' of 'str' objects}
     4    0.013    0.003    0.013    0.003 {method 'search' of '_sre.SRE_Pattern' objects}
 28284    0.012    0.000    0.012    0.000 {method 'seek' of 'cStringIO.StringO' objects}
    10    0.011    0.001    0.297    0.030 {method 'join' of 'str' objects}
 14141    0.010    0.000    0.014    0.000 client.py:1685(pipeline_execute_command)
 14142    0.010    0.000    0.010    0.000 _compat.py:22(<lambda>)
     2    0.007    0.004    0.007    0.004 {method 'sendall' of '_socket.socket' objects}
```

Nous voyons que le script `skyrock.py` s'est exécuté en 1,34 secondes, que la génération des UUID a demandé environ une demi-seconde et l'exécution du pipeline, 650 millisecondes. Plutôt pas mal pour l'insertion de 14 141 UUID en tout dans Redis avec un langage client interprété. Bien sûr, nous sommes en local, donc le réseau n'entre pas en ligne de compte.

Ce petit exercice nous permet de démontrer la simplicité et l'efficacité de l'utilisation de Redis pour ce genre de besoins. En réalité, Skyrock utilise une méthode juste un peu plus compliquée, mais vous voyez que la rapidité de Redis le permet. En effet, les clés maintenues par minute dans Redis accumulent les utilisateurs qui ont envoyé une requête durant cette minute. Sur un site web, il est bien sûr impossible (sans passer par de l'Ajax, par exemple) de savoir si le navigateur de l'utilisateur est toujours ouvert sur une page, donc s'il est toujours connecté. Pour savoir si un ami est en ligne, le code de Skyrock utilise cette règle : si on trouve un identifiant d'ami dans les cinq dernières minutes, cela veut dire qu'il est en ligne. En Redis, cela veut dire qu'il faut effectuer une union des sets des cinq dernières clés, à l'aide de l'instruction `SUNION`, puis faire l'intersection avec le résultat. Dans les dernières versions de Redis, cela pourrait être réalisé du côté du serveur avec un script en langage Lua.

Contourner les problèmes

En 2010, en mettant cette solution en place avec Redis, les développeurs de Skyrock se sont heurtés à un problème inhérent au client PHP : comme il n'y avait pas encore de support du *pipelining* comme nous l'avons utilisé dans notre script en Python, la création du set temporaire incluant la liste d'amis se révélait peu performante, uniquement à cause du client PHP. Leur solution a été de développer l'envoi en Redis en utilisant les fonctions d'accès direct au réseau de PHP, en créant des sockets avec `fsockopen()` et en envoyant ainsi les requêtes au serveur Redis, ce qui a permis de profiter de toutes les fonctionnalités de Redis. En d'autres termes, pour pallier les limitations

du driver PHP, ils ont développé le leur afin d'intégrer les fonctionnalités de pipelining et de pouvoir ajouter plusieurs membres à un ensemble en une seule commande (la commande SADD permet d'ajouter plusieurs éléments à un ensemble depuis Redis 2.4). Comme pour les outils Topy et Fluxy, Skyrock est logé à la même enseigne que beaucoup de sociétés du web qui utilisent le NoSQL : ces technologies sont jeunes et tout n'y est pas encore parfait. En revanche, elles sont libres, ouvertes et souples, il ne reste donc plus qu'à développer des outils ou des améliorations pour adapter les moteurs NoSQL aux besoins. Un exemple de cette démarche est le contournement de l'absence d'index secondaires. Dans plusieurs moteurs NoSQL, nous l'avons vu, l'accès se fait exclusivement, ou en tout cas de préférence, par la clé. La recherche à l'intérieur de la structure de la valeur est souvent pénalisante, voire impraticable. Pour pallier ce type de contrainte, beaucoup d'utilisateurs de moteurs NoSQL dupliquent les données ou créent d'autres ensembles de paires clé-valeur où la nouvelle clé est un élément extrait d'une valeur des informations d'origine, sur laquelle on souhaite chercher, et la nouvelle valeur est l'identifiant par lequel on va pouvoir retrouver toute la structure dans la collection d'origine, c'est-à-dire sa clé.

Utiliser Redis pour le traitement d'images

Redis est un moteur convaincant. Lorsqu'on l'a essayé et qu'on a pu constater sa rapidité, sa simplicité et sa puissance, on commence à lui trouver de nombreuses utilisations. Redis a été mis en place chez Skyrock pour un autre besoin encore que ceux évoqués précédemment : le traitement des images déposées par les utilisateurs, avec un module nommé WIR *(Web Image Reworker)*. Lorsqu'un utilisateur dépose une image sur le site skyrock.com, il peut la redimensionner et procéder à un certain nombre de transformations, par exemple lui appliquer un filtre. Les images originales sont déposées sur un cluster de stockage. Afin d'accélérer le traitement et d'éviter de saturer les I/O sur le cluster de stockage, les images sont maintenues en mémoire quelques temps sur les machines de traitement dans une base Redis, qui stocke les métadonnées des images et le fichier lui-même dans son état original. Comme il s'agit d'un groupe de machines de traitement, les images sont réparties sur les machines en utilisant un algorithme de *consistent hashing* sur l'URI du fichier. Lorsque le site doit afficher une image, il reconnaît sur quelle machine de traitement doit se trouver l'image en recréant le hachage à partir de l'URI du fichier. Il fait donc une demande à la machine de traitement, qui sert l'image depuis la base Redis si elle y est présente, ou qui la récupère depuis le cluster de stockage et l'insère dans Redis si elle n'y est pas encore. Répétons que c'est toujours l'image originale qui est stockée dans Redis. Le traitement est effectué dynamiquement en local sur la machine de traitement par rapport à la définition du traitement choisie par l'utilisateur pour cette image, et renvoyée au serveur post-traitement, basé sur nginx qui va conserver dans un cache disque l'image transformée.

Les applications mobiles : Smax

Skyrock est également présent sur les smartphones, avec un site optimisé pour les terminaux mobiles (skyrock.mobi) et des applications disponibles pour IOS et Android. Outre l'application Skyrock pour écouter la radio, une application de chat est disponible, nommée Smax.

Smax permet de chatter, d'envoyer des messages à une liste d'amis et de contacter d'autres utilisateurs de Smax situés à proximité, grâce à son support de la géolocalisation.

Smax est développé en Node.JS et utilise MongoDB comme moteur principal de stockage. C'est un choix naturel pour une application géolocalisée grâce à son support des données géospatiales. MongoDB implémente en effet une indexation géospatiale à deux dimensions qui permet d'effectuer des recherches sur des documents contenant des coordonnées de type X et Y ou longitude et latitude.

Exemple de requêtes géographiques dans MongoDB

Illustrons à travers un exemple l'utilisation de MongoDB comme moteur SIG (Système d'information géographique) simplifié (puisqu'il stocke pour l'instant uniquement des points, pas des polygones). Dans l'invite interactive de MongoDB, nous créons les documents suivants dans une collection que nous nommerons `monuments` :

```
db.monuments.save({ «_id»: «eiffel», «loc»: {«lon»:2.29510, «lat»:48.857910}, «nom»:
«Tour Eiffel»});
db.monuments.save({ «_id»: «liberty», «loc»: {«lon»:-74.0447, «lat»:40.6894}, «nom»:
«Statue de la Liberté»});
```

Nous incluons dans notre document un objet nommé `loc`, composé de deux coordonnées. Nous créons ensuite un index géographique sur cet objet :

```
db.monuments.ensureIndex({«loc»: «2d»});
```

Nous allons maintenant essayer les opérateurs de recherches géographiques de MongoDB :

```
db.monuments.find({ loc: { $nearSphere: [2,48] }});
db.monuments.find({ loc: { $nearSphere: [-74,48] }});
```

Nous utilisons ici l'opérateur `$nearSphere` pour retourner une liste de documents ordonnée par leur proximité avec les coordonnées fournies. L'opérateur `$near` existe également, mais il effectue la comparaison par rapport à un système de références plat. `$nearSphere` effectue le calcul en se basant sur une géométrie sphérique. Les résultats obtenus sont illustrés sur la figure 16-8.

Figure 16-8
Requêtes géographiques dans MongoDB

```
> db.monuments.find({ loc: { $nearSphere: [2,48] }});
{ "_id" : "eiffel", "loc" : { "lon" : 2.2951, "lat" : 48.85791 }, "nom" : "Tour Eiffel" }
{ "_id" : "liberty", "loc" : { "lon" : -74.0447, "lat" : 40.6894 }, "nom" : "Statue de la Libert
é" }
> db.monuments.find({ loc: { $nearSphere: [-74,48] }});
{ "_id" : "liberty", "loc" : { "lon" : -74.0447, "lat" : 40.6894 }, "nom" : "Statue de la Libert
é" }
{ "_id" : "eiffel", "loc" : { "lon" : 2.2951, "lat" : 48.85791 }, "nom" : "Tour Eiffel" }
>
```

Nous voyons que les documents sont retournés dans des ordres différents, en restituant en premier les points les plus proches de la coordonnée fournie. Pour trouver la distance de nos coordonnées avec un emplacement, nous pouvons utiliser la commande `geoNear` :

```
db.runCommand( { geoNear: "monuments", near: [2, 48], spherical: true, num: 1,
distanceMultiplier: 6378000 });
```

qui retourne un document JSON indiquant la distance du document le plus proche des coordonnées fournies. Un seul document est retourné car nous avons demandé un seul résultat (`num: 1`). Le document retourné est le suivant :

```
{
        «ns»: «test.monuments»,
        «near»: «1100010000000110001011001010010000000110001011001010»,
        "results": [
                {
                        "dis": 97955.81154374767,
                        "obj": {
                                "_id": "eiffel",
                                «loc»: {
                                        «lon»: 2.2951,
                                        «lat»: 48.85791
                                },
                                «nom»: «Tour Eiffel»
                        }
                }
        ],
        "stats": {
                "time": 1,
                "btreelocs": 0,
                "nscanned": 2,
                "objectsLoaded": 2,
                «avgDistance»: 97955.81154374767,
                «maxDistance»: 0.015358595804450191
        },
        «ok»: 1
}
```

Voilà pour ce qui concerne l'utilisation des outils NoSQL chez Skyrock.

Conclusion : comment se présente le futur ?

Au début des années 1980, les systèmes de gestion de bases de données relationnelles (SGBDR) furent une technologie de rupture. Ce terme, « technologie de rupture » est un concept introduit par Clayton M. Christensen, professeur de *business* à l'université Harvard, dans son livre *The Innovator's Dilemma*. L'idée du professeur Christensen est que, principalement dans les innovations techniques, il arrive qu'une technologie établie soit remplacée par une technologie naissante qui évolue à une très grande vitesse et qui prend la technologie établie par surprise. L'innovation peut être inférieure technologiquement au modèle établi, mais plus simple et plus accessible. Au fur et à mesure de son développement, elle comble son retard technologique et finit par remplacer l'ancienne technologie qui n'a pas su s'adapter. Inversement, l'innovation peut être supérieure, mais incomprise du grand public car trop innovante justement. Certaines sociétés peuvent s'en emparer, travailler à son développement sans grand écho au début, et finalement surpasser la technologie existante au point de la tuer. Le modèle relationnel fut à l'époque un exemple d'innovation de rupture supérieure.

Le mouvement NoSQL est-il une technologie de rupture, qui prend le modèle traditionnel du SGBDR par surprise ? C'est en tout cas ce que disent en filigrane beaucoup de ceux qui créent ou utilisent des bases NoSQL. Et s'il s'agit d'une technologie de rupture, est-ce une rupture inférieure ou supérieure ? Dans ce livre, nous avons parlé technique – car c'est souvent ce qui intéresse en premier lieu les informaticiens –, mais nous avons aussi essayé d'élargir le débat et d'aborder des conceptions plus théoriques. En effet, nous pensons que l'informatique a le droit aux concepts et à la réflexion. Les approches sont parfois trop pragmatiques, sans aucune recherche ou intérêt sur la raison pour laquelle les choses sont faites ainsi. Le risque ici est que le modèle relationnel ou les différents modèles NoSQL soient utilisés sans réflexion, et pour les seconds, uniquement pour suivre le mouvement. Le modèle relationnel peut se targuer d'une forme de supériorité conceptuelle, car il provient d'une réflexion brillante et offre de grandes qualités d'abstraction et de représentation des données. Cependant, ses défenseurs font souvent l'impasse sur ses limitations : la difficulté de représenter efficacement tous les types de données et toutes les formes d'organisation, le niveau de compétence qu'il demande, sa sensibilité à la qualité de la modélisation préliminaire des données (c'est-à-dire sa dépendance envers les fondations) et sa difficulté – en tout cas dans ses implémentations les plus classiques, car ce défaut n'est en rien dû au modèle

qui justement s'abstrait de l'implémentation physique – à se distribuer sur un grand nombre de machines. En d'autres termes, il s'agit d'un modèle hégémonique, alors que les besoins sont divers.

Parallèlement, il existe un autre mouvement dont on parle un peu et qui a été nommé NewSQL. Il s'agit de concevoir des systèmes distribués qui utilisent le modèle relationnel et respectent l'ACIDité de la transaction. Des moteurs comme NuoDB ou VoltDB sont des moteurs dits NewSQL. NuoDB, auparavant NimbusDB, est un projet de moteur relationnel dans le cloud qui est en version béta et qui approche d'une livraison finale. VoltDB est un projet cofondé par Michael Stonebreaker, chercheur important dans le monde des bases de données, puisqu'il est à l'origine des moteurs Ingres et PostgreSQL et ancien *Chief Technical Officer* (CTO) d'Informix. VoltDB est sorti en version 1.0 en mai 2010 et évolue en deux éditions : une *community edition* libre sous licence GPL v3, et une édition entreprise commerciale.

Récemment, le modèle relationnel a retrouvé un certain brillant. En 2012, Google a publié un papier de recherche (http://research.google.com/pubs/pub38125.html) présentant leur nouvelle base de données, nommée F1. Elle est basée sur une technologie présentée comme étant le successeur de BigTable (dont nous avons pu mesurer l'importance pour le monde NoSQL dans ce livre), nommée Google Spanner. Il s'agit d'un moteur relationnel distribué et transactionnel, qui se base sur Paxos et qui n'utilise pas de système de fichiers distribué. La réplication est basée sur un horodatage rendu possible par l'utilisation sur les machines qui forment le grid de modules GPS ou de modules de communication avec des horloges atomiques pour la synchronisation des horloges.

Par rapport à ces tendances, on pourrait résumer l'approche NoSQL par cet adage d'Adam Osborne, un des pionniers de l'ordinateur personnel : « adequacy is sufficient ». Si vous n'avez pas besoin de transactions et de données fortement structurées, et que vous voulez pouvoir monter en charge par la distribution des données à moindre coût sur des machines standards, vous voilà armé pour faire votre choix et pour commencer à explorer le moteur qui vous convient le mieux.

Index

A

Amazon 15
arbre de Merkle 88
architecture distribuée 71
Avro 65

B

bases
  orientées colonnes 117
  orientées documents 114
Big Data 13, 49, 118
BigTable 76, 90, 117, 225, 263
BSON (Binary Serialiazed dOcument Notation ou Binary JSON) 186

C

Cassandra 16, 64, 84, 93, 225, 257, 276, 283, 291, 299
  cassandra-cli 227
  CQL 228, 270
  nodetool 232
CFEngine 279
clé 250, 264, 266, 269
Cloudera 129
Cobol (COmmon Business Oriented Language) 8
Codasyl (Conference on Data Systems Languages) 8
Codd, Edgar Frank 9, 13
  douze règles de Codd 10
cohérence 267
cohérence finale 87
Conception pilotée par le domaine 266
Couchbase Server 261
CouchDB 37, 70, 89, 161, 237, 239, 257, 258
  CouchApp 265
curl 67, 165
Cypher 244

D

Datastax 23
défaut d<0092>impédance objet-relationnel 34
Dynamo 15, 77, 225
DynamoDB 17

E

ElasticSearch 235, 250
Erlang 162

F

Facebook 16
famille de colonnes 142, 225

G

Ganglia 301
GeoJSON 255
gestion de versions 89
Google 13, 59
  BigTable 15
  GFS 14

H

hachage consistant 78

Hadoop 14, 53, 129
HBase 15, 49, 142, 256
HDFS 131
hinted handoff 230

I

iftop 295
index 268

J

Java 134
JSON 250
JSON (JavaScript Object Notation) 68, 115

L

Lisp 99

M

MapReduce 44, 98, 253
memcached 211
Mémoire 271
modèle
  de données réseau 8
  hiérarchique 8
  relationnel 9, 31, 32, 38, 118
  Waterfall 29
MongoDB 36, 48, 58, 72, 107, 185, 253, 258, 268, 270, 276, 280, 298
  GridFS 188
  réplication 197
  sharding 199
  SIG 315

N

NULL 118

O

OLAP (Online Analytical Processing) 12, 50
OLTP (Online Transactional Processing) 11, 50
Oracle 22, 53
ORM (Object-Relational Mapping) 36

P

paire clé-valeur 110
paires clé-valeur 250
pidstat 296
Protocol Buffers 59
protocole
  de bavardage 77

R

Redis 56, 81, 211, 260, 269, 272, 281, 290, 300, 308, 314
  redis-benchmark 219
  redis-cli 214
  types de données 212
REST (Representational State Transfer) 37, 165, 195
Riak 94, 203, 259, 277, 281

S

sar 297
schema-less 30, 107
sharding 31, 75
SpiderMonkey 189
SSH 131
sysstat 293
systèmes de gestion de bases de données
  modèle de données réseau 8
  modèle relationnel 9
  OLAP 12
  OLTP 11

T

théorème CAP 43, 87
  PACELC 46
Thrift 61, 147
tombstone 143
transaction 39, 47
  distribuée 42
transactions 188

V
Vector clocks 93
Virtualisation 277

W
WAL (Write-Ahead Logging) 47, 226
Write-Ahead Logging 47

X
XML 115, 264, 285
  E4X 265